André Luiz do Vale Soares

Verificação Formal de Aplicações para Reconhecimento de Ondas Eletrocardiográficas

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

01 de Março de 2002

Resumo

Este trabalho apresenta uma metodologia de verificação formal aplicada ao projeto de uma aplicação biomédica. Esta aplicação faz o reconhecimento de dados obtidos a partir do eletrocardiograma de pacientes cardíacos, em um sistema de acompanhamento de tempo real desses pacientes.

A aplicação utiliza autômatos finitos determinísticos para identificar não somente as ondas eletrocardiográficas como também a forma como estas ondas são apresentadas durante o exame, emitindo diagnósticos e iniciando as ações necessárias quando o paciente encontrar-se em situações de risco.

Neste contexto, a metodologia proposta, que envolve a verificação formal do mecanismo de reconhecimento de ondas eletrocardiográficas e seqüências de ondas com a utilização da ferramenta de verificação de modelos *Verus*, tem apresentado resultados concretos na correção e projeto deste mecanismo de reconhecimento através da identificação e correção de falhas, tornando o sistema mais seguro e confiável.

A aplicação da metodologia permitiu que outras ondas fossem reconhecidas pelo mecanismo de leitura de dados eletrocardiográficos, de forma que fosse feito o re-projeto do autômato para reconhecimento de ondas, bem como fosse projetado um autômato para o reconhecimento de seqüências de ondas, contribuindo de maneira concreta para o desenvolvimento da aplicação biomédica.

Abstract

This work presents a formal verification methodology applied to a biomedical application design for data recognizing which are obtained on electrocardiogram in a real time monitoring system.

The application uses deterministic finite automata to identify electrocardiographic waves and the form which they are disposed in the exam, diagnosing and starting the needed actions when the patient is on risk.

In this way, the proposed methodology involves the formal verification of electrocardiographic waves and waves sequences recognizing system by the *Verus* model checker, and have shown good results in the correction and design of this mechanism, identifying some faults and giving more reliability to the system.

The application of the methodology has permitted to recognize other waves in the electrocardiographic module, re-designing the actual wave recognizing automata and designing the waves sequence recognizing automata, leading to a real contribution in the biomedical system implementation.

Agradecimentos

Ao finalizar este trabalho, após dois anos e seis meses, tive o prazer de contar com a amizade e o incentivo de pessoas que tornaram mais suave este caminho. A elas, agradeço:

Aos meus pais e meu irmão, que ficaram privados de minha companhia diversas vezes, mas sempre me incentivaram e apoioaram.

À minha "família"em Belo Horizonte, Adeildo e Ernande, com quem pude dividir não somente a moradia, mas também alegrias e angústias durante o tempo em que ficamos em Minas Gerais.

Aos meus companheiros Tayana, Tanara, Tânia, Altigran, Eduardo Souto, Pinheiro, Daniela, Pio, Márcia, Rosana, Eduardo Nakamura, Maurício, Arlindo e Gilbert, que me acompanharam durante esta fase.

À Paula que, mesmo distante, deu-me companhia e incentivo para prosseguir.

Aos colegas da FUCAPI, Jackson, Aninha, Brasílio, Pedro, Leonardo e Miranda, que permitiram minha ida à Belo Horizonte sem comprometer o trabalho por mim desempenhado na fundação.

À toda a diretoria e gerência da FUCAPI, Dra. Isa, Dr. Folhadela, Niomar, Guajarino, Dimas e Wilson, pela oportunidade de realizar este mestrado.

À Renata, Emília e todos os funcionários e professores da UFMG e da UA, com quem sempre pude contar em Minas Gerais.

Ao Prof. Sérgio, Edjard, Autran e Claudionor, pelo apoio, ajuda e paciência fundamentais para a realização deste trabalho.

Ao Hervaldo, pela parceria e confiança depositadas no trabalho.

E, sobretudo, a Deus, por ter dado a força necessária para superar todas as dificuldades surgidas neste caminho.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Programas

Capítulo 1

Introdução

Sistemas críticos de tempo real têm sido utilizados em diversas aplicações, tais como: indústria, controle de tráfego aéreo, controle de tráfego terrestre, sistemas de controle para plantas de energia, controle de armamentos, monitoramento de pacientes, entre outros.

Esses sistemas caracterizam-se por terem que produzir seus resultados em intervalos de tempos específicos. Além disto, há pouca ou, em alguns casos, nenhuma tolerância a falhas, especialmente quando de seu correto funcionamento dependem vidas humanas.

Para assegurar o correto funcionamento de um sistema crítico de tempo real podem ser utilizadas algumas abordagens bastante conhecidas, como simulação de sistemas e *model checking*. Em ambas, contudo, é necessário construir um modelo, quer para simulação, quer para verificação, para então obter informações sobre as propriedades do sistema quando da mudança do tempo.

1.1 Contexto

Neste sentido, tem sido desenvolvida uma aplicação biomédica crítica de tempo real, para o acompanhamento de pacientes cardíacos [Car00].

Este sistema, chamado LIFE GUARD, baseia-se na tecnologia de *wearable computers*, analisando o funcionamento do coração através do eletrocardiograma do paciente, obtendo um diagnóstico e comunicando-se com uma central, quando necessário, através de uma

rede sem fio.

Temos então um tipo de aplicação onde dois módulos essenciais são encontrados. O primeiro refere-se à comunicação entre o equipamento instalado no paciente e a central receptora. Caso haja alguma falha neste processo, todo a funcionalidade do sistema torna-se inútil diante do risco de vida que corre o paciente. O segundo módulo diz respeito ao tratamento das informações fornecidas pelo eletrocardiógrafo do equipamento. Deve-se assegurar não só o correto diagnóstico ante as informações colhidas, como também uma correta leitura do eletrocardiograma.

Uma das principais funcionalidades do sistema reside no reconhecimento das ondas produzidas por um eletrocardiógrafo, como também de seqüências. Isto é feito por meio de dois autômatos finitos determinísticos. Um para o reconhecimento das ondas produzidas e outro para o reconhecimento de seqüências obtidas com a geração destas ondas.

Desta forma, é imprescindível para que o sistema seja confiável a correta leitura e interpretação dos sinais eletrocardiográficos. Falhas neste mecanismo poderiam trazer sérios problemas ao paciente, podendo inclusive implicar em risco de morte.

1.2 Motivação

Técnicas de Verificação Formal têm sido aplicadas com êxito em diversos tipos de aplicações de tempo real, inclusive em aplicações biomédicas [CCMM95].

O reconhecimento das ondas eletrocardiográficas é um fator crítico para o funcionamento do sistema. Caso alguma onda que reflita uma situação real de perigo de vida por que passe o paciente não seja reconhecida, não é somente a funcionalidade de um sistema que está comprometida, mas sim uma vida humana. Portanto a utilização de técnicas formais eficientes para validar o módulo de reconhecimento de ondas pode contribuir de forma essencial para que seja garantido um alto grau confiança exigido por um sistema deste porte.

1.3 Objetivos

Este trabalho tem como objetivo o emprego de uma metodologia baseada em Verificação Formal para a correção e proposta de um mecanismo de reconhecimento de ondas eletrocardiográficas e seqüências de ondas baseado em autômatos finitos determinísticos.

A metodologia proposta envolve a utilização do verificador de modelos *Verus*, baseado na técnica de verficação chamada *Symbolic Model Checking*.

Nesta direção, a aplicação da metodologia tem dado uma grande contribuição à aplicação biomédica de reconhecimento de sinais eletrocardiográficos, através da correção e proposta tanto do autômato de reconhecimento de ondas quanto do autômato de reconhecimento de seqüências, através da identificação de ondas e seqüências ainda não reconhecidas.

Com isto, o emprego desta metodologia de verificação formal apresenta resultados concretos no projeto de uma aplicação biomédica de acompanhamento de pacientes, fazendo com que esta aplicação atinja um maior grau de segurança e confiança.

1.4 Organização da Dissertação

Este trabalho está dividido em seis seções.

A primeira seção (capítulo 2) contém trabalhos relacionados, encontrados na literatura, com o trabalho aqui desenvolvido e apresentado.

A segunda seção (capítulo 3) descreve o sistema LIFE GUARD e discorre sobre ondas eletrocardiográficas, apresentando também o autômato utilizado no sistema para o reconhecimento de ondas.

A terceira seção (capítulo 4) apresenta a técnica de Verificação Formal através de Model Checking, descrevendo seus fundamentos e principais características, como a utilização de BDDs e Lógica Temporal, citando também algumas aplicações úteis desta técnica.

A quarta seção (capítulo 5) apresenta a ferramenta utilizada para a modelagem do problema, *Verus*, além de outras ferramentas também baseadas em SMC *Symbolic Model Checking*.

A quinta seção (capítulo 6) descreve os modelos utilizados para a verificação formal das

máquinas de estado.

A sexta e última seção (capítulo 7) contém as conclusões obtidas a partir do trabalho realizado, bem como aponta trabalhos futuros para o mesmo tema.

Capítulo 2

Verificação Formal com Model Checking

2.1 Model Checking

Model Checking caracteriza-se por ser uma técnica de verificação de sistemas concorrentes de estados finitos [CGP01]. O método têm sido empregado com êxito na verificação de projetos de circuitos complexos, como também na verificação do funcionamento de protocolos de comunicação. Sua principal característica é o tratamento com problemas de explosão no espaço de estados. Isto ocorre quando os componentes de um sistema podem interagir com outros, ou quando os sistemas possuem estruturas que podem assumir muitos valores diferentes. Em tais casos, o número de estados globais do sistema pode ser muito grande.

Os principais métodos para validação de sistemas complexos são simulação, testes, verificação dedutiva e *model checking*. Em se tratando de *software*, tanto simulação quanto testes implicam em fornecer determinados dados de entrada para observar qual a saída produzida. A principal diferença é que a simulação é feita a partir de uma abstração ou de um modelo do sistema, enquanto o teste é realizado na própria aplicação. Estes métodos são bastante eficientes para encontrar erros. Contudo, sua utilização para verificar todas as possíveis interações e potenciais "armadilhas"é raramente possível.

A verificação dedutiva utiliza axiomas e regras de prova para checar o grau de correção de sistemas. Sua importância é altamente reconhecida por cientistas de computação [CGP01]. Entretanto, caracteriza-se por ser um processo que consome bastante tempo e requer peritos e/ou especialistas que tenham bastante experiência em raciocínio lógico. Sua utilização na prova de correção de um sistema pode levar vários dias ou meses.

Já a técnica de *Model Checking* ou **Verificação de Modelo** caracteriza-se por aplicar-se a sistemas concorrentes de estados finitos. É uma técnica onde a verificação de um sistema pode ser feita de forma automática, pois todo o processo é realizado sobre as propriedades pertinentes ao sistema, por meio de uma pesquisa exaustiva no espaço de estados dessas propriedades.

De forma simplificada, a técnica consiste em construir um modelo de estados do sistema a ser testado, onde as variáveis a serem testadas podem assumir valores lógicos ou inteiros. São definidos então processos nesse modelo que podem alterar os valores dessas variáveis. Por fim, são atribuídos valores às variáveis dos modelos, conforme as condições definidas em cada processo. Utilizando Lógica Temporal, verifica-se então as propriedades mais relevantes do modelo do sistema. Assim sendo, temos então três passos principais na aplicação da técnica [CGP01]:

Modelagem Consiste na conversão do sistema em um modelo que será utilizado em uma ferramenta de verificação.

Especificação Consiste em definir as propriedades do modelo que deverão ser verificadas pela ferramenta. É comum a utilização de Lógica Temporal para definir tais propriedades.

Verificação Em geral é feita de forma automática por meio de ferramentas destinadas a isto. Nesta fase as propriedades definidas na especificação são testadas e um valor lógico é então atribuído a cada propriedade. No caso de propriedades receberem o valor *falso*, é necessário prover um **contra-exemplo**, a fim de que modificações necessárias sejam efetuadas no modelo.

2.2 Fundamentos para Verificação com *Model Checking*

Ao modelar sistemas, deve-se extrair as propriedades que efetivamente podem e devem ser utilizadas para verificar o quão correto é o sistema.

Podemos observar que o sistema em estudo possui a característica de sistema reativo [Sun98], por necessitar de maneira freqüente de interação com o ambiente do qual faz parte, sem a garantia de finalizar sua execução. Podemos citar como aplicações de sistemas reativos, entre outros:

- Sistemas de Controle de Aeronaves
- Sistemas Operacionais

Em ambos os sistemas, a computação ocorre de forma que o computador reage indefinidamente aos estímulos do ambiente, produzindo, desta forma, um efeito no mundo físico, e não somente um valor computacional.

Para verificar a corretude de um sistema, devemos primeiro especificar as propriedades que o sistema deveria possuir. Em seguida, devemos construir um *modelo formal* do sistema, capturando as propriedades especificadas.

Em se tratando de sistemas reativos, a primeira característica que devemos capturar são os **estados** que o sistema pode assumir [CGP01]. Um estado é uma espécie de "fotografia"do sistema em um determinado momento, e é dado pelos valores que as variáveis representando as propriedades do sistema assumem em um dado instante. Entre a passagem de um estado para outro de um sistema reativo, dizemos que ocorre uma **transição**. Dessa forma, toda a computação acerca de sistemas reativos dá-se pela obtenção de suas transições, onde os valores das variáveis de um estado seguinte são determinados pelos valores dessas variáveis em um estado imediatamente anterior.

Para o cômputo dessas transições, é utilizada uma estrutura baseada em transição de grafos chamada **estrutura de Kripke** (2.3).

Da mesma maneira, as propriedades acerca dos estados dos modelos e suas relações entre as transições - ou seja, a especificação do modelo - é descrita de forma apropriada através de CTL* (2.6), uma forma estendida de lógica temporal (2.4).

2.3 Estrutura de Kripke

A rigor, uma estrutura de Kripke pode ser definida como uma máquina de estado finito não determinística, cujos estados são classificados com variáveis booleanas, que são resultados de avaliações de expressões naquele estado.

Seja AP um conjunto de proposições atômicas . Uma estrutura de *Kripke* M sobre AP é uma quádrupla $M = (S, S_0, R, L)$, onde [CGP01]:

1. S é um conjunto finito de estados.
2. $S_0 \subseteq S$ é o conjunto de estados iniciais.
3. $R \subseteq S \times S$ é uma relação de transição que deve ser total, isto é, $\forall s \in S, \exists t \in S, (s,t) \in R$.
4. $L : S \rightarrow 2^{AP}$ é uma função que associa cada estado ao conjunto de proposições atômicas verdadeiras naquele estado.

Podemos encontrar variantes desta definição considerando apenas o conjunto finito de estados, a relação de transição e a função de associação.

Em se tratando de sistemas reativos, toda a sua computação pode ser definida por meio de transições, onde o que se observa é uma seqüência infinita de estados, e cada estado é obtida de um estado anterior por alguma transição. A estrutura de *Kripke* serve para descrever esse comportamento.

2.4 Lógica Temporal

Lógica Temporal [Vis98] é uma linguagem de especificação formal para utilização em verificadores de modelos (*model checkers*). É, na verdade, uma lógica estendida com modalidades temporais, o que permite a especificação de ordens de eventos no decorrer do tempo, sem a necessidade de introduzir explicitamente o "tempo". Possui operadores temporais, conforme descrito na seção 2.6, que permitem formular sentenças acerca de modelos, do tipo: "para todos os futuros momentos em que p for verdadeiro, haverá um momento

futuro em que q será verdadeiro". Enquanto a lógica tradicional permite especificar propriedades relacionadas aos estados iniciais e finais de sistemas finitos, a lógica temporal é mais adequada para descrever o comportamento dinâmico de sistemas reativos.

2.5 Symbolic Model Checking

Em um algoritmo de *Symbolic Model Checking* a relação de transição é representada implicitamente por fórmulas booleanas, e os estados não são explicitamente enumerados [CC94].

As especificações são feitas utilizando-se fórmulas baseadas em CTL (seção 2.6). Estas fórmulas são implementadas através de BDDs (seção 2.7), permitindo a verificação de sistema relativamente grandes e com muitas transições de estados e variáveis.

2.6 Computation Tree Logics

2.6.1 Sintaxe

CTL é uma lógica temporal *branching − time*, cuja interpretação está associada a estados de sistemas de transição. Utiliza uma linguagem proposicional [Gor00] com um conjunto de proposições atômicas (variáveis proposicionais) AP, operadores temporais **X** (próxima vez) e **U** (até que), além de **R** (Release), e quantificadores de caminho $\forall$, significando "para todos os caminhos", e $\exists$, significando "para algum caminho", que pode ser considerado como uma abreviação de $\neg\forall\neg$.

Dois tipos de fórmulas são tradicionalmente introduzidas por CTL*: fórmulas de estado e fórmulas de caminho. Esta distinção é essencial para algumas de suas versões restritas, como CTL, mas também é conveniente para a verdadeira definição: as primeiras são avaliadas em estados e as outras em caminhos no modelo. Aqui há uma definição recursiva conjunta de fórmulas de estado e de caminho:

- Toda proposição atômica é uma fórmula de estado;
- Se ϕ,ψ são fórmulas de estado, então também são $\neg\phi$ e $\phi \wedge \psi$;

- Se ϕ é uma fórmula de caminho, então $\forall\phi$ é uma fórmula de estado;
- Toda fórmula de estado é uma fórmula de caminho;
- Se ϕ,ψ são fórmulas de caminho, então também são $\neg\phi$,$\phi \rightarrow \psi$,$\mathbf{X}\phi$ e $\phi\mathbf{U}\psi$;

Podemos então combinar todas estas definições em uma definição uniforme de fórmula de CTL*, denominada forma de Backus-Naur(BNF):

$$\phi := p \mid \neg\phi \mid \phi_1 \rightarrow \phi_2 \mid \mathbf{X}\phi \mid \phi_1\mathbf{U}\phi_2 \mid \forall\phi \tag{2.1}$$

onde p é uma proposição atômica.
Dois dos mais populares predecessores de CTL* são:

- UB (introduzida por Ben-Ari, Manna e Pnuelli, em 1981), a linguagem que contém os operadores temporais **X**, **G** (sempre no futuro), com o dual **F** (eventualmente), mas não **U**. A definição BNF da fórmula de UB é:

$$\phi := p \mid \neg\phi \mid \phi_1 \rightarrow \phi_2 \mid \exists\mathbf{X}\phi \mid \exists\mathbf{G}\phi \mid \forall\mathbf{G}\phi \tag{2.2}$$

- CTL (introduzida por E. Clarke e Emerson, em 1981), com a mesma linguagem de CTL e a definição BNF de fórmula:

$$\phi := p \mid \neg\phi \mid \phi_1 \rightarrow \phi_2 \mid \exists\mathbf{X}\phi \mid \exists(\phi_1\mathbf{U}\phi_2) \mid \forall(\phi_1\mathbf{U}\phi_2) \tag{2.3}$$

2.6.2 Semântica de CTL*

Modelos de CTL* são árvores computacionais geradas por execuções não determinísticas de programas concorrentes. Consistem em todos os caminhos computacionais possíveis de um determinado programa, ou seja, nas seqüências de estados consecutivos numa execução de um programa.

A noção de semântica básica para CTL* é *verdadeiro em um estado s de um modelo M*, definido por indução simultânea nas fórmulas de estado e caminho como segue:

1. $M \models_s p$, se $s \in V(p)$;
2. $M \models_s \neg\phi$, se $not M \models_s \phi$;
3. $M \models_s \phi \rightarrow \psi$, se $M \models_s \phi$ implica em $M \models_s \psi$;
4. $M \models_s \forall\phi$, onde ϕ é uma fórmula de caminho, se para todo caminho p começando de s, $M \models_p \phi$.

Nas cláusulas seguintes p é um caminho $\{p_0, p_1, ...\}$ e p^i é o caminho sufixo $\{p_i, p_{i+1}, ...\}$.

1. $M \models_p \phi$, onde ϕ é uma fórmula de estado, se $M \models_{p_0} \phi$;
2. $M \models_p \neg\phi$, se $not M \models_p \phi$;
3. $M \models_p \phi \rightarrow \psi$, se $M \models_p \phi$ implica em $M \models_p \psi$;
4. $M \models_p \mathbf{X}\phi$, se $M \models_{p^i} \phi$;
5. $M \models_p \phi \mathbf{U} \psi$, se há $i \geq 0$, tal que $M \models_{p^i} \psi$ e para todo j, tal que $0 \leq j < i$, $M \models_{p^j} \phi$.

Uma fórmula de estado ϕ é válida em um modelo M, denotado por $M \models \phi$, se ϕ é verdadeira em todo estado de M. Uma fórmula de estado ϕ é CTL*-válida se for válida em todo o modelo CTL*.

2.6.3 CTL e LTL

Computation Tree Logic (CTL) é um subconjunto de CTL* [CGP01]. Consiste basicamente em que cada ocorrência de um operador de caminho deve ser imediatamente precedida por um quantificador de caminho. Já Lógica Temporal Linear ou *Linear Temporal Logic*(LTL) consiste em fórmulas na forma **AP**, onde as únicas subfórmulas de estado de **P** são proposições.

Ao escrever especificações temporais a serem utilizadas por um verificador de modelos, muitos fatores devem ser considerados [Vis98]. Mas o primeiro e principal diz respeito à capacidade da lógica temporal disponível no verificador. Por exemplo, se uma propriedade a ser verificado puder ser expressa tanto em LTL quanto em CTL, é preferível utilizar LTL

devido às suas fórmulas serem mais sucintas e mais legíveis que as fórmulas CTL. Uma razão para isto é que essas fórmulas não requerem quantificadores de caminho desnecessários antes de todos os operadores temporais.

Contudo, um argumento que vai contra o uso de LTL é que seu algoritmo de verificação de modelo é de complexidade exponencial ao tamanho da fórmula [CES83]. Infelizmente, escrever uma fórmula equivalente em CTL pode não ser a solução. Como exemplo, consideremos a fórmula $A(G_{p_o} \vee G_{p_1} \vee ... \vee G_{p_n})$, que possui uma fórmula CTL equivalente exponencial em n. Além disto, se a fórmula LTL original não puder ser expressa em CTL, então escrevê-la em LTL é a única opção.

Conforme [CGP01], há dez operadores CTL básicos:

- **AX** e **EX**,
- **AF** e **EF**,
- **AG** e **EG**,
- **AU** e **EU**,
- **AR** e **ER**.

Cada um dos dez operadores pode ser expresso em termos de três operadores **EX**, **EG** e **EU**:

- $\mathbf{AX}f = \neg\mathbf{EX}(\neg f)$
- $\mathbf{EF}f = \mathbf{E}[True\ \mathbf{U}\ f]$
- $\mathbf{AG}f = \neg\mathbf{EF}(\neg f)$
- $\mathbf{AF}f = \neg\mathbf{EG}(\neg f)$
- $\mathbf{A}[f\ \mathbf{U}\ g] \equiv \neg\mathbf{E}[\neg g\ \mathbf{U}\ (\neg f \wedge \neg g)] \wedge \neg\mathbf{EG}\neg g$
- $\mathbf{A}[f\ \mathbf{R}\ g] \equiv \neg\mathbf{E}[\neg f\ \mathbf{U}\ \neg g]$
- $\mathbf{E}[f\ \mathbf{R}\ g] \equiv \neg\mathbf{A}[\neg f\ \mathbf{U}\ \neg g]$

Podemos citar, como exemplo, algumas fórmulas CTL [CGP01]:

- $\mathbf{EG}(Start \wedge \neg Ready)$: É possível alcançar um estado onde *Start* ocorra mas *Ready* não.
- $\mathbf{AG}(Req \rightarrow \mathbf{AF}Ack)$: Se uma requisição ocorrer, então ela será eventualmente reconhecida.
- $\mathbf{AG}(\mathbf{EF}Restart)$: De qualquer estado é possível alcançar o estado *Restart*.

2.6.4 Exemplo do Forno de Microondas

Utilizaremos como exemplo para modelagem utilizando CTL o funcionamento de um forno de microondas [CGP01]. A estrutura de Kripke é descrita na figura 2.1.

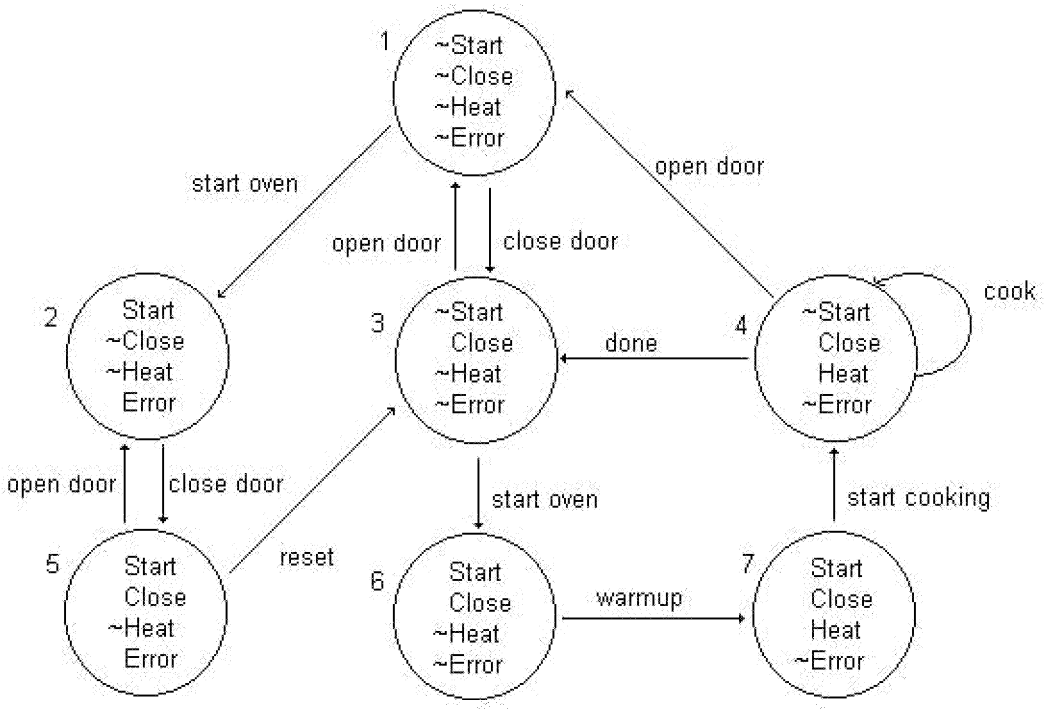

Figura 2.1: Estrutura de Kripke para o funcionamento de um forno de microondas

O objetivo é verificar a satisfabilidade da fórmula CTL $\mathbf{AG}(Start \rightarrow \mathbf{AF}Heat)$ (Se o forno for ligado, então ele ficará eventualmente aquecido), que é equivalente à fórmula $\neg\mathbf{EF}(Start \vee \mathbf{EG}\neg Heat)$.

Inicialmente devemos verificar o conjunto de estados que satisfaça as fórmulas atômicas e depois as subfórmulas mais complicadas. Utilizaremos $S(g)$ para denotar o conjunto inicial de todos os estados em que a subfórmula g é *verdadeira*. Assim Temos que:

- $S(Start) = 2, 5, 6, 7$
- $S(\neg Heat) = 1, 2, 3, 5, 6$

Para calcular $S(\mathbf{EG}\neg Heat)$ devemos primeiramente encontrar o conjunto de componentes fortemente conexos não triviais de $S' = S(\neg Heat)$.

Um Componente Fortemente Conexo ou *Strongly Connected Component* (SCC) de um grafo dirigido $G = (V, E)$ pode ser definido da seguinte maneira [CBBL91] : O conjunto de vértices V pode ser particionado em classes equivalentes $V_i, 1 \leq i \leq r$, tal que vértices (ou nós) v e w são equivalentes se e somente se existem caminhos de v para w e de w para v. Seja $E_i, 1 \leq i \leq r$ o conjunto de arcos conectando os vértices em V_i. Então os grafos $G_i = (V_i, E_i)$ são chamados **Componentes Fortemente Conexos** (SCC) de G. Um SCC trivial é aquele em que há um simples vértice, enquanto um SCC não trivial possui mais de um vértice. Assim, se eliminarmos qualquer nó de um SCC não trivial de um grafo dirigido, teremos eliminado pelo menos um ciclo do grafo.

Desta forma, ao analisarmos a figura, vemos que o SCC de S' é o conjunto $\{\{1, 2, 3, 5\}\}$. Procedemos agora nomeando T, que é o conjunto de todos os estados que deveriam ser classificados por $\mathbf{EG}\neg Heat$ para ser a união sobre os elementos de SCC, que é inicialmente $T = \{1, 2, 3, 5\}$. Nenhum outro estado em S' pode alcançar um estado em T por todo o caminho de S'. Assim, temos:

- $S(\mathbf{EG}\neg Heat) = \{1, 2, 3, 4, 5\}$

Agora podemos calcular:

- $S(Start \wedge \mathbf{EG}\neg Heat) = \{2, 5\}$

Ao calcular $S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat))$, devemos começar com $T = S(Start \wedge \mathbf{EG}\neg Heat)$. Em seguida, utilizamos a conversão da relação de transição para classificar todos os estados aonde a fórmula é satisfeita. Assim, temos:

- $S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat)) = \{1,2,3,4,5,6,7\}$
- $S(\neg\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat)) = \emptyset$

Desde que o estado inicial 1 não esteja contido neste conjunto, podemos concluir que o sistema descrito pela estrutura de Kripke não satisfaz a especificação dada.

2.7 *Reduced Ordered Binary Decision Diagram*

Binary Decision Diagrams ou Diagramas Binários de Decisão ou simplesmente BDDs [MCC00] constituem-se em uma maneira eficiente para a representação de fórmulas binárias. Trabalhos publicados por Randal Bryant [Bry86] e Fabio Somenzi [Som99] têm mostrado que BDDs reduzidos e ordenados (ROBDDs) são representações canônicas de funções booleanas. Isto significa que duas fórmulas booleanas são logicamente equivalentes se e somente se seus BDDs são isomorfos. Isto simplifica a execução de operações que são executadas freqüentemente, como verificação de equivalência de duas fórmulas ou decisão se uma dada fórmula é ou não satisfatível [CGP01, MCC00].

Um BDD é um grafo dirigido acíclico $(V \cup \{0,1\})$, onde $v \in V$ representa o conjunto de variáveis de uma fórmula booleana F. Sua estrutura compreende $n-2$ vértices qualificados com elementos de V, onde cada vértice possui 2 arcos que dele incidem, e dois vértices folha, qualificados com os valores lógicos *verdadeiro* e $falso$ (0,1), possuindo dois arcos incidentes e nenhum arco que deles parta.

Em termos práticos, a representação de fórmulas booleanas através de BDDs ordenados diminui de forma considerável o tamanho do modelo a ser checado, se comparada a formas mais clássicas, como árvores de decisão, especialmente quando tratamos de representações para relações de transições com muitos estados. Como exemplo, podemos ver na figura 2.2 uma comparação entre a árvore de decisão e o BDD para a fórmula booleana $(a \wedge b) \vee (c \wedge d)$.

Desta forma, ferramentas baseadas em *Symbolic Model Checking*, como o Verus por exemplo, utilizam ROBDDs para a avaliação de suas fórmulas, de forma a reduzir o tamanho do modelo, permitindo assim a sua avaliação através de uma estrutura bem reduzida.

Através da árvore de decisão, podemos verificar que os estados $(ab\overline{cd})$,$(ab\overline{c}d)$,$(abc\overline{d})$,$(abcd)$ satisfazem a fórmula dada. Entretanto, podemos perce-

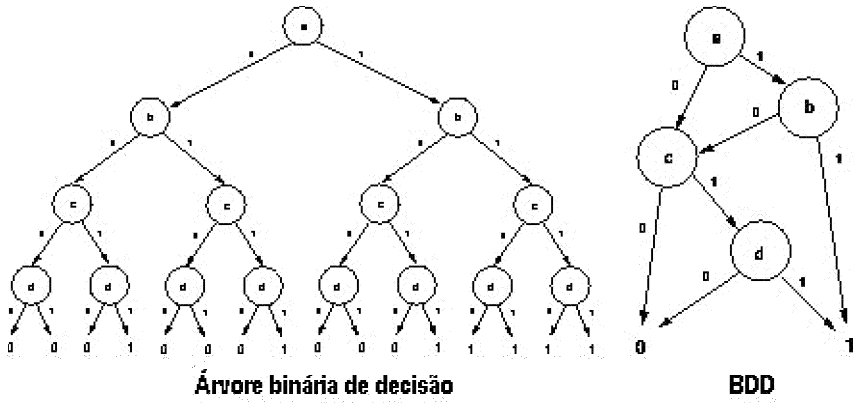

Figura 2.2: Árvore de decisão e BDD para a fórmula $(a \wedge b) \vee (c \wedge d)$

ber que sempre que a e b forem tiverem o valor *verdadeiro*, a fórmula será satisfatível, ou seja, o estado (ab) é suficiente para chegar a um estado final *verdadeiro*, conforme verificado no BDD da mesma fórmula booleana. Seguindo o mesmo raciocínio, podemos verificar que o estado (cd) também torna a fórmula satisfatível, embora tal estado não seja verificado neste BDD. Contudo, se o primeiro vértice do BDD representasse a variável booleana c, chegaríamos a este estado.

O importante aqui é observar que o BDD reduz consideravelmente a quantidade de estados possíveis para satisfazer a fórmula booleana, o que em tese diminui o tempo de computação para verificar a satisfabilidade de um modelo. Chega-se aos mesmos resultados que os fornecidos por uma árvore de decisão, por exemplo, porém com muito menos caminhos a serem percorridos.

Desta forma, BDDs constituem-se em uma boa abordagem para ferramentas de verificação de modelos de sistemas reativos, uma vez que permite um melhor tratamento computacional para estas aplicações cuja principal característica é justamente a explosão de estados possíveis a serem considerados.

2.7.1 Ordenação de Variáveis

O tamanho de um BDD gerado para uma determinada função depende da ordem escolhida para as variáveis. Há funções, como soma de produtos, onde cada produto tem apoio disjunto do outro, para as quais algumas ordenações levam a BDDs que são lineares

no número de variáveis, enquanto outras ordenações produzem nós que são exponenciais no número de variáveis.

O método exato mais simples para encontrar a ordenação ótima de variáveis é tentar encontrar todas as ordenações possíveis. Para n variáveis, existem $n!$ ordens diferentes, de forma que o custo para a construção do BDD é exponencial em n no pior caso. Conseqüentemente, um método que utilize a abordagem de "força-bruta"para a construção do BDD requer tempo $O(n!2^n)$.

Este tempo pode ser comprovado através da observação de que duas permutações de variáveis que possuem o mesmo sufixo em comum gerarão dois BDDs com partes mais baixa idênticas [Som99].

Suponhamos que f é uma função lógica de n variáveis $x_1, ..., x_n$. Seja $N = \{1, ..., n\}$ o conjunto de índices das variáveis. Uma ordem π de $x_1, ..., x_n$ é uma permutação de N. O BDD para f abaixo da ordem π é denotado por $BDD(f, \pi)$. Seja B um subconjunto de N e $\Pi(B)$ o conjunto de ordens cujos últimos membros $|B|$ pertençam a B, ou seja:

$$\Pi(B) = \{\pi | \pi[n - j + 1] \in B, j = 1, ..., |B|\}. \tag{2.4}$$

Nós devemos nos referir àquelas variáveis cujos índices estão em B como as variáveis de fundo de $\Pi(B)$. As variáveis cujos índices estão em $N - B$ serão as variáveis de topo. Também, para uma ordem π de uma variável x_i, definimos $N_i(f, \pi)$ como o número de nós x_i em $BDD(f, \pi)$.

Reordenação Dinâmica

Técnicas que melhoram iterativamente ordenação de variáveis são freqüentemente empregadas "dinamicamente": se o tamanho dos BDDs crescem a partir de um limiar durante a execução de uma operação, a própria operação é suspensa e a reordenação é então efetuada, usualmente por meio de alguma variação de filtragem. Após a reordenação, a operação que foi interrompida é geralmente reiniciada. Esta abordagem dinâmica tem mostrado bons efeitos em muitas aplicações [Som99], pois permite que a reordenação intervenha sempre que se tornar necessária. Por outro lado, em verificação seqüencial não é incomum que a maior parte do tempo seja tomada pela tarefa de reordenação, de forma que aparente-

mente um certo nível de controle do processo através da aplicação seja benéfico. Escrever funções de manipulação de forma que elas possam ser interrompidas de maneira segura pela reordenação requer um certo cuidado. Os detalhes dependem da implementação do pacote de BDDs.

Capítulo 3

O Sistema LIFE GUARD

Doenças cardiovasculares e respiratórias têm sido causas freqüentes de óbitos em diversos países desenvolvidos [Car00]. Muitas fatalidades poderiam ser evitadas através de um diagnóstico preventivo e um acompanhamento contínuo das atividades cardíacas do paciente.

Eventos cardio-respiratórios tidos como causa de morbidade e mortalidade são mais relacionados à arritmias cardíacas, deterioração hemodinâmica ou queda na concentração de oxigênio no sangue. Desta forma, eventos como doenças cardíacas e cerebrovasculares, além de males respiratórios crônicos, são responsáveis por um grande número de tratamentos hospitalares, alto custo de tratamentos de reabilitação e alto grau de abstenção diária no trabalho.

O alto custo dos serviços hospitalares e o aumento da preocupação com qualidade de vida em serviços de saúde têm atraído um grande interesse e melhoria em tratamentos domiciliares. Além disso, monitoramentos clínicos durante atividades diárias são muito restritos ao monitoramento feito com *holter*. Este tipo de equipamento tem processamento e interpretação *off-line* e, conseqüentemente, nenhuma atuação.

3.1 Descrição do Sistema

Há quatro importantes sinais vitais que podem representar a função cardio-respiratória: Atividade Elétrica Cardíaca, Pressão Sangüínea Sistêmica, Concentração de Oxigênio no

Sangue e Fluxo Sangüíneo [Car00]. Eles podem ser medidos por técnicas invasivas e não invasivas. As abordagens não invasivas comumente utilizam Eletrocardiograma (ECG), Medição Indireta de Pressão Sangüínea, Oximetria de Pulso e *Doppler* vascular. O eletrocardiograma é capaz de mostrar distúrbios na taxa e o ritmo cardíaco. Além disto, baseado em diversas formas de atividade elétrica, pode fornecer inferências sobre anormalidades cardíacas anatômicas e funcionais. Pressão sangüínea e avaliação de *Doppler* vascular representam a hemodinâmica cardiovascular. Oximetria de Pulso é influenciada pela função respiratória, concentração de hemoglobina e fluxo sangüíneo. Estes exames parecem ser suficientes para detectar quase todos os eventos cardio-respiratórios súbitos e para acompanhar pacientes de alto risco fora de ambientes hospitalares.

Neste sentido, tem sido desenvolvido o sistema LIFE GUARD [Car00], que vem a ser um monitor de sinais vitais móvel, com capacidade de atuação. O sistema baseia-se na captura de sinais, processamento e módulo de conectividade através de um computador usável (*wearable computer*); o monitoramento inteligente e o controlador de atuação são baseados em *software* e *hardware* (FPGA) acoplados ao módulo do computador. O monitoramento central é outra parte do sistema. O monitoramento inteligente e o sistema de atuação são localizados tanto no computador quanto no módulo central. Assim, pode-se obter um controle tanto local quanto à distância dos eventos. A rede possui uma conexão sem fio (*Wavelan* e *Bluetooth*) e uma conexão via modem.

Uma das partes mais importantes do sistema diz respeito ao reconhecimento de ondas geradas durante um eletrocardiograma [Car00]. O sistema utiliza um algoritmo de identificação de ondas contínuas, baseado na teoria de autômatos agregada a sistemas especialistas e redes neurais. O algoritmo foi desenvolvido baseado em critérios clínicos de diferenciação e identificação de ondas: forma, duração, amplitude, presença de transições abruptas ou baixas de ondas e seqüências de ondas identificadas. Tais parâmetros foram utilizados em diversas formas de apresentação de ondas eletrocardiográficas, detalhadas na seção 3.2. O aspecto da mudança de direção determina se a transição é abrupta ou baixa. A seqüência de ondas identificadas e suas características são dinamicamente comparadas para detectar alterações nas ondas. Esta comparação é feita em direções simultâneas e é muito importante para a análise contínua.

3.2 Ondas Eletrocardiográficas

Um eletrocardiograma (ECG ou EKG) representa a atividade elétrica das células de um coração normal. Este impulso é gerado por um pequeno grupo de células conhecido como *nodo sinusal* ou *nódulo de Keith-Flach*. Este nodo encontra-se localizado na parte superior da aurícula direita, na desembocadura da veia cava superior. Este grupo de células é o principal marcapasso do coração, devido à sua capacidade de produzir um maior número de despolarizações por minuto, que varia entre 90 e 60 [Gar].

O estímulo se propaga por todo o miocárdio auricular, produzindo sua contração. Posteriormente, este estímulo alcança a união atrioventricular, que está, por sua vez, envolvida pelo tecido automático (nodo de *Aschoff-Tawara*) e pelo tecido de condução (*His*). Daqui surgem dois ramos: um à esquerda e outro à direita, por onde o estímulo elétrico de distribui por ambos os ventrículos através do *Sistema de Purkinje*.

Esta transmissão do impulso elétrico através das células miocárdicas é que dará lugar às diversas ondas que aparecem em um eletrocardiograma. Em uma amostra comum de um eletrocardiograma, podemos identificar quatro tipos distintos de ondas e três tipos de intervalos ou segmentos.

O primeiro tipo é chamado de onda **P**, e representa a despolarização atrial, ou seja, o tempo necessário para um impulso elétrico do nodo sinoatrial (SA) dilatar-se por toda a musculatura atrial. Sua amplitude não deve exceder 2 a 2.5 mm de altura e sua duração é de 0.06 a 0.11 segundos. Normalmente, uma onda P precede um complexo de ondas QRS.

Entre o início de uma onda P e o início de um complexo QRS encontramos um intervalo ou segmento de onda chamado **P-R**. Este segmento representa o período de inatividade elétrica correspondente ao atraso fisiológico que um estímulo leva em todo o nodo atrioventricular. Sua duração compreende 0.12 a 0.20 segundos.

Após um segmento de onda P-R, encontramos um complexo de ondas chamado **QRS**, que compreende na verdade três tipos de ondas: Q, R e S. O complexo QRS representa a despolarização ventricular. A onda Q é sempre encontrada no início do complexo e pode ou não estar presente no mesmo. A onda R é sempre a primeira deflexão positiva. A onda S representa a deflexão negativa e sempre ocorre após uma onda R. A amplitude do complexo QRS varia conforme a idade e o sexo do paciente e sua duração não ultrapassa 0.10 segundos.

Do início de um complexo de ondas QRS ao final de uma onda T podemos encontrar um segmento de ondas **Q-T**. Este intervalo representa o tempo necessário para a despolarização e repolarização ventricular. Sua duração varia conforme a idade, o sexo e a freqüência cardíaca do paciente, geralmente variando entre 0.32 e 0.40 segundos.

A onda **T** representa a repolarização dos ventrículos. Em ocasiões raras, podemos também encontrar um tipo de onda denominado de onda **U**, que reflete a repolarização das fibras de *His-Purkinje*.

O segmento de ondas **S-T** representa o final de uma despolarização ventricular e o início de uma repolarização dos ventrículos. Este segmento compreende o final de uma onda S e o início de uma onda T. Sua duração geralmente não é medida.

Na figura 3.1 podemos observar a seqüência normal obtida a partir de um eletrocardiograma.

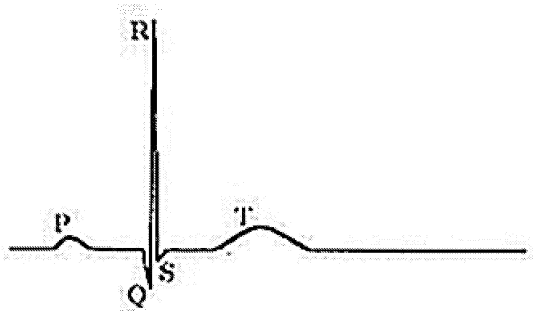

Figura 3.1: Tipos de Ondas Geradas por um Eletrocardiógrafo

3.2.1 Seqüências de Ondas

Em um eletrocardiograma normal, a seqüência comum pode ser observada na figura 3.2.

Normalmente, seqüências são iniciadas por ondas P. Em seguida, é esperado um segmento PR, seguido de uma complexo QRS. Em seguida, normalmente a seqüência é fechada por um segmento ST, seguido de uma onda T. Em casos raros, pode-se observar uma onda U após a onda T [JG98, Flo].

Contudo, as seqüências podem perder suas características normais, em virtude de diversos fatores, como ruídos durante a obtenção do eletrocardiograma, por exemplo.

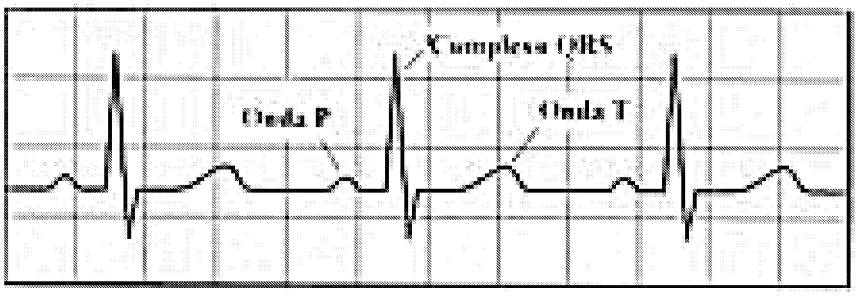

Figura 3.2: Seqüência de Ondas

Os fatores ligados à presença de disfunções no ritmo cardíaco são os mais preocupantes e merecem maior atenção no estabelecimento de diagnósticos rápidos e ações mais eficazes. E o desafio para um sistema de acompanhamento de pacientes está justamente no reconhecimento de situações anormais, que impliquem em algum risco de morte ao paciente. Quanto mais anomalias forem detectadas, e as ações necessárias forem tomadas em tempo hábil, mais confiável é o sistema.

Em um eletrocardiograma de um paciente com ritmo de escape juncional ou com ritmo juncional acelerado, por exemplo, a onda P pode estar invertida, ausente na seqüência ou pode ocorrer somente após um complexo QRS [Yan]. Isto poderia gerar seqüências diferentes, que devem ser detectadas pelo sistema.

Em uma fibrilação atrial, o ritmo cardíaco é irregular e o que se observa durante o eletrocardiograma é uma onda P praticamente ausente, sendo observado, contudo, a ocorrência normal do complexo QRS, do segmento ST e da onda T [dEdC].

Já em uma fibrilação ventricular, os impulsos elétricos são disparados por múltiplos pontos do ventrículo, de forma rápida e desordenada. Assim, o ventrículo "treme"e não consegue ter uma contração efetiva, cessando o bombeamento de sangue. É uma situação parecida com uma parada cardíaca e, a menos que um ritmo efetivo seja restaurado dentro de poucos minutos, poderá levar o paciente à morte. No eletrocardiograma, o que se observa é um traçado completamente desordenado, sem a inscrição de complexos QRS [dEdC].

Um segmento S-T com curta duração pode levar à leitura de uma seqüência onde não há a clara distinção entre o final de um complexo QRS e o início de uma onda T. Neste caso, o complexo QRS, o segmento S-T e a onda T poderiam ser lidos como um complexo QRS, que seria então seguido de uma onda P, indicando o reinício da seqüência, ou possivelmente

de uma onda U.

De qualquer forma, a correta leitura da seqüência de ondas geradas pelo eletrocardiograma é fundamental para o diagnóstico do paciente. A diversidade de seqüências identificadas permite um melhor e mais correto diagnóstico da situação do paciente, provendo ao médico informações mais precisas para que as ações corretas sejam tomadas.

3.3 Reconhecimento de Ondas e Seqüências de Ondas

3.3.1 Autômatos Finitos

Autômatos finitos são reconhecedores de linguagens regulares [NMDB99].

Um autômato finito tem um conjunto de *estados*, alguns dos quais são denominados *estados finais*. À medida que caracteres da *string* de entrada são lidos, o controle da máquina passa de um estado a outro, segundo um conjunto de *regras de transição* especificadas para o autômato. Se após o último caracter o autômato encontra-se em um dos estados finais, a *string* foi reconhecida (ou seja, pertence à linguagem). Caso contrário, a *string* não pertence à linguagem aceita pelo autômato.

Formalmente , autômatos finitos são reconhecedores de linguagens regulares, definidos através de quíntuplas da forma: $M = (E, V, f, q_0, F)$, onde :

E é um conjunto finito não vazio de *estados* do autômato finito.

V é denominado *alfabeto de entrada* do autômato e corresponde a um conjunto finito não vazio dos *símbolos de entrada* ou *átomos* indivisíveis que compõem a cadeia de entrada submetida ao autômato para aceitação.

f é uma função de transição de estados do autômato e seu papel é o de indicar transduções possíveis em cada configuração do autômato. Esta função mapeia o produto cartesiano $E \times (V \cup \{\lambda\})$ em E, ou seja, fornece para cada par (*estado, símbolo de entrada*) um novo estado para onde o autômato deverá mover-se.

q_0 é denominado *estado inicial* do autômato finito, e corresponde a um elemento do conjunto E. É o estado para o qual o reconhecedor deve ser levado antes de iniciar suas atividades ($q \in V$).

F é um subconjunto do conjunto E dos estados do autômato, e contém todos os *estados de aceitação* ou *estados finais* do autômato finito. Estes estados são aqueles em que o autômato deve terminar o reconhecimento das cadeias de entrada que pertencem à linguagem que o autômato define. Nenhuma outra cadeia deve ser capaz de levar o autômato a qualquer destes estados ($F \subseteq V$).

Como exemplo, suponhamos a seguinte quíntupla: $M = (\{A, B\}, \{0, 1\}, f, A, \{B\})$, onde:

$$f = \begin{array}{l} (A, 0) \Rightarrow A \\ (A, 1) \Rightarrow B \\ (B, 0) \Rightarrow A \\ (B, 1) \Rightarrow B \end{array}$$

Para este autômato finito, reconhecem-se os seguintes elementos:

- estados do autômato: A e B
- símbolos do alfabeto de entrada: 0 e 1
- estado final: B
- estado inicial: A
- linguagem reconhecida: cadeias de dígitos binários terminadas obrigatoriamente por um dígito 1.

Desta forma, podemos classificar os autômatos finitos como determinísticos ou não determinísticos. Um autômato finito determinístico é aquele em que, dado um estado e uma transição, leva a um único estado, enquanto o autômato finito não determinístico pode levar a mais de um estado.

Têm-se empregado autômatos em diversas aplicações da área biomédica, com maior concentração em Modelagem da Atividade Elétrica Cardíaca, Redes do Sistema Imunológico, Modelagem de Substratos Cardíacos Isquêmicos Arritmogênicos, entre outras [Car00].

Na área de reconhecimento de sinais eletrocardiográficos, Koski publicou um trabalho sobre sua modelagem, mas não de sua identificação. Diversos trabalhos apresentam a utilização de *matching de templates* das ondas do eletrocardiograma baseado em autômatos.

Florenz publicou um trabalho sobre a utilização de uma máquina de estados finitos para um rápido processamento eletrocardiográfico . Udupa utilizou a análise sintática para análise do ritmo em eletrocardiogramas . Hu e colaboradores descreveram a utilização de uma mistura de sistemas especialistas para classificar as ondas do eletrocardiograma.

No LIFE GUARD são utilizados dois autômatos finitos determinísticos, sendo um para o reconhecimento de ondas eletrocardiográficas e o outro para o reconhecimento de seqüências de ondas. Nesta seção será descrito o autômato para reconhecimento de ondas. O projeto e formalização do autômato de reconhecimento de seqüências de ondas faz parte do trabalho a ser realizado a partir da aplicação da metodologia proposta para verificação deste mecanismo.

O autômato de reconhecimento de ondas possui 59 estados finais aceitáveis, um estado inicial e 93 estados intermediários, divididos em cinco grupos ou níveis de transição. Contudo, o autômato pode comportar até 80 estados finais e 160 estados intermediários.

Foram definidos cinco níveis de transição no autômato para modelar ondas eletrocardiográficas. O primeiro nível representa uma queda de tensão abaixo de 10mV (milivolts). O segundo representa uma queda de tensão entre 1 e 10mV. O terceiro representa ausência de tensão (0 mV). O quarto nível representa subida de tensão entre 1 e 10 mV, enquanto o quinto e último nível representa subidas de tensão acima de 10 mV.

É importante ressaltar que uma onda ocorre de maneira contínua, sendo essa divisão em cinco níveis criada a partir de uma abstração que possibilite a montagem do autômato para implementação de um reconhecedor de ondas, ou seja, foi feita a discretização da representação dos sinais elétricos, sem contudo implicar em uma perda da característica da onda obtida através da emissão em seqüência destes sinais.

Tomemos como exemplo uma onda obtida a partir da ocorrência de uma subida de tensão para 3 mV. Podemos distinguir duas mudanças de estados quando do mapeamento desta onda para um AFD. A primeira mudança ocorre quando saímos da faixa onde o nível de tensão é de cerca de 0 mV para uma faixa superior seguinte, onde a tensão fica em torno de 1 mV. Por fim, ocorre a queda de tensão para 0 mV, onde temos a segunda transição.

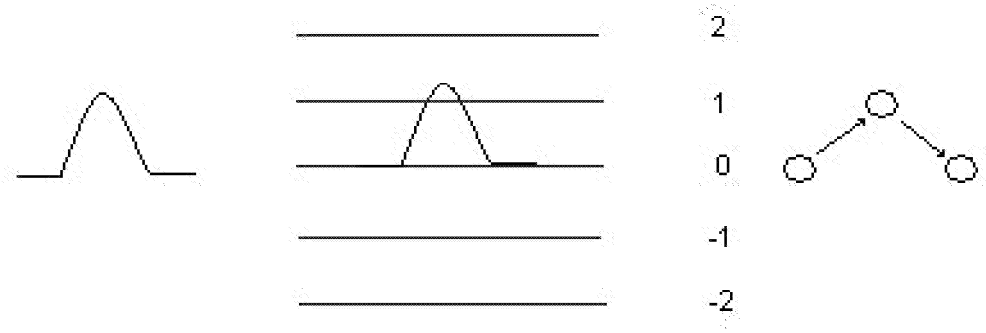

Figura 3.3: Exemplo de mapeamento de uma onda para um autômato finito determinístico

Formalização

Formalmente, o AFD do Life Guard pode ser descrito como uma quíntupla $M = (E, V, f, q_0, F)$, onde :

E é um conjunto finito não vazio de estados do autômato finito, que corresponde ao subconjunto de números inteiros {0..225}.

V é o alfabeto de entrada do autômato, compreendendo o subconjunto de números inteiros {-1..2}.

f é uma função de transição de estados do autômato: $f(E, V) \rightarrow E$.

q_0 é o estado inicial do autômato, formado pelo número inteiro 0.

F é um subconjunto do conjunto E dos estados do autômato, formado pelo subconjunto de números naturais {1..59}.

Ou: $M = (\{0..225\}, \{-1..2\}, f, 0, \{1..59\})$

3.3.2 A tabela de transições de estado

Uma tabela de transição é uma matriz na qual as colunas representam os estados do autômato e as linhas os símbolos do alfabeto. Cada entrada na matriz indica qual o estado final de uma transição a partir do estado indicado na coluna, através do símbolo indicado na linha. Assim, colunas da tabela de transição representam o estado corrente, e as linhas, o símbolo corrente. Uma entrada na tabela (cruzamento entre linha e coluna) pode ser vazia, indicando que não há transição possível, ou ter o próximo estado que é atingido pela transição corrente.

Desta forma, podemos definir um autômato finito determinístico como sendo aquele em que para cada combinação de estado e entrada da tabela de transição de estados existe uma única transição aplicável.

Assim, Hervaldo Sampaio [Car01] utiliza autômatos no módulo de reconhecimento de ondas e seqüências, lidas a partir de dados coletados por um eletrocardiógrafo junto ao paciente. Através deste autômato, Carvalho, que é médico cardiologista, previu o reconhecimento de 59 ondas em seu autômato.

O reconhecimento de ondas também é feito por um autômato. Porém sua descrição e formalização é um dos objetivos deste trabalho.

O autômato finito determinístico de reconhecimento de ondas do LIFE GUARD é representado através da tabela de transições de estados 3.1. Conforme esta tabela, pode-se verificar que o autômato possui 59 estados finais, podendo comportar até 80. São utilizados também 32 estados para o nível 1, embora seja possível utilizar até 50 estados para este nível. Da mesma forma, são utilizados 19 estados no nível 2, sendo permitido utilizar até 30 estados, 27 estados no nível -1, permitindo-se até 50 e 15 estados no nível -2, sendo permitidos até 30 estados.

Partindo-se da premissa de que todo o processo de obtenção e digitalização das ondas é correto, não cabendo mecanismos de verificação do mesmo, o que interessa então é verificar e assegurar que quaisquer ondas possíveis de serem observadas durante um exame eletrocardiográfico sejam reconhecidas por este autômato.

estado	0	1	2	-1	-2	**estado**	0	1	2	-1	-2	**estado**	0	1	2	-1	-2
0	0	81	131	174	218	112	55	112	-	182	224	173	26	96	140	173	-
81	1	81	131	161	211	131	17	88	131	170	216	174	33	100	143	174	218
82	5	82	132	164	213	132	8	83	132	166	214	175	34	111	149	175	222
83	8	83	133	166	214	133	11	84	133	168	215	176	37	103	145	176	219
84	11	84	-	168	215	134	15	86	134	-	-	177	38	-	-	177	-
85	14	85	134	-	-	135	16	87	135	-	-	178	39	105	146	178	220
86	15	86	135	-	-	136	18	89	136	172	217	179	40	107	147	179	-
87	16	87	-	-	-	137	22	91	137	-	-	180	42	-	-	180	221
88	17	88	136	170	216	138	24	93	138	-	-	181	43	-	-	181	-
89	18	89	-	172	217	139	28	95	139	-	-	182	56	-	-	182	224
90	21	90	137	-	-	140	30	97	140	-	-	183	57	-	-	183	-
91	22	91	141	-	-	141	31	98	141	-	-	184	58	-	-	184	225
92	23	92	138	-	-	142	32	99	142	-	-	185	59	-	-	185	-
93	24	93	142	-	-	143	36	101	143	178	220	186	48	109	148	186	-
94	27	94	139	-	-	144	41	102	144	180	221	187	53	-	-	187	-
95	28	95	-	-	-	145	45	104	145	-	-	211	3	85	134	162	211
96	29	96	140	-	-	146	47	106	146	-	-	212	4	-	-	163	212
97	30	97	-	-	-	147	50	108	147	-	-	213	7	-	-	165	213
98	31	98	-	-	-	148	52	110	148	-	-	214	10	-	-	167	214
99	32	99	-	-	-	149	55	112	149	182	224	215	13	-	-	169	215
100	35	100	143	176	219	161	2	82	132	161	211	216	20	92	138	171	216
101	36	101	144	178	220	162	3	85	134	162	212	217	26	96	140	173	217
102	41	102	-	180	221	163	4	-	-	163	-	218	34	111	149	175	218
103	44	103	145	-	-	164	6	-	-	164	213	219	38	-	-	177	219
104	45	104	-	-	-	165	7	-	-	165	-	220	40	107	147	179	220
105	46	105	146	-	-	166	9	-	-	166	214	221	43	-	-	181	221
106	47	106	-	-	-	167	10	-	-	167	-	222	48	109	148	186	222
107	49	107	147	-	-	168	12	-	-	168	215	223	53	-	-	187	223
108	50	108	-	-	-	169	13	-	-	169	-	224	57	-	-	183	224
109	51	109	148	-	-	170	19	90	137	170	216	225	59	-	-	185	225
110	52	110	-	-	-	171	20	92	138	171	223						
111	54	111	149	184	225	172	25	94	139	172	217						

Tabela 3.1: Tabela de Transição de Estados do AFD para Reconhecimento de Ondas

Capítulo 4

Ferramentas para Verificação de Modelos

Existem várias ferramentas baseadas em *Symbolic Model Checking* que permitem a verificação de modelos. Entre elas, podemos citar: Verus, SPIN, STeP e SMV, cujas principais características serão mostradas a seguir.

4.1 Verus

4.1.1 Descrição

Verus é uma ferramenta para especificação formal e verificação de sistemas críticos em tempo real ou não [SEM97, CC01].

Sua sintaxe é parecida com a sintaxe de comandos da linguagem C. Contudo, permite apenas o uso de dois tipos de variáveis: inteiros com tamanho fixo e lógicos. O sistema a ser verificado é especificado através da linguagem do Verus, para ser então compilado em um grafo de transição de estado. Algoritmos derivados de *Symbolic Model Checking* são utilizados pra calcular informação quantitativa sobre o modelo. Um *symbolic model checker* baseado em CTL também é fornecido para acompanhar a verificação.

O suporte a não determinismo é realizado através de uma instrução *select*, que é o

mecanismo que permite ao projetista modelar todas as possibilidades de ocorrência de uma determinada variável.

A especificação de propriedades temporais do modelo dá-se através de fórmulas CTL (propriedades com tempo não determinado) e RTCTL (propriedades expressas com tempo determinado). O principal mecanismo de implementação de unidades de tempo no modelo é feito através da instrução *wait()*. A cada instrução *wait()* no modelo, um novo estado é então tomado, ou seja, todas as alterações efetuadas após o *wait()* imediatamente anterior são então efetuadas. Podemos visualizar isto através de um grafo denominado *Grafo de Espera* ou *Wait Graph*. Cada transição deste grafo corresponde a uma passagem de uma unidade de tempo no modelo (4.1).

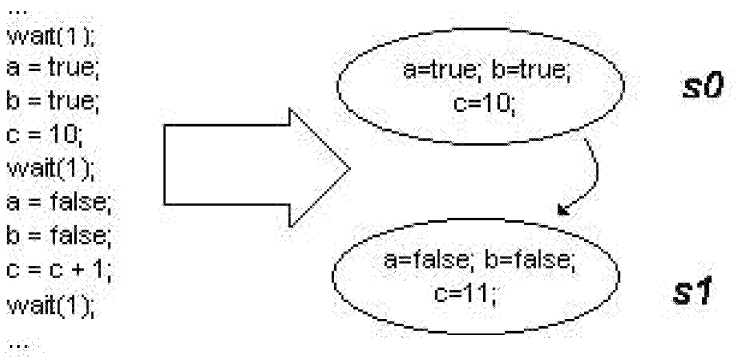

Figura 4.1: Exemplo de *Wait Graph*

A linguagem de especificação do *Verus* foi projetada para permitir uma descrição clara e concisa das características temporais dos programas. A informação produzida permite ao usuário verificar o comportamento temporal do modelo. A escalabilidade das tarefas do sistema pode ser determinada calculando-se seus tempos de resposta. Tempos de reação a eventos e muitos outros parâmetros do sistema podem ser analisados por este método. Esta informação também provê uma percepção acerca do comportamento do sistema e em muitos casos pode auxiliar na identificação de ineficiências e sugerir otimizações no projeto. Os mesmos algoritmos podem então ser utilizados para análise de desempenho do projeto modificado. A avaliação de como a otimização afeta o projeto pode ser feita antes que a modificação seja implementada. Assim pode-se reduzir custos de desenvolvimento.

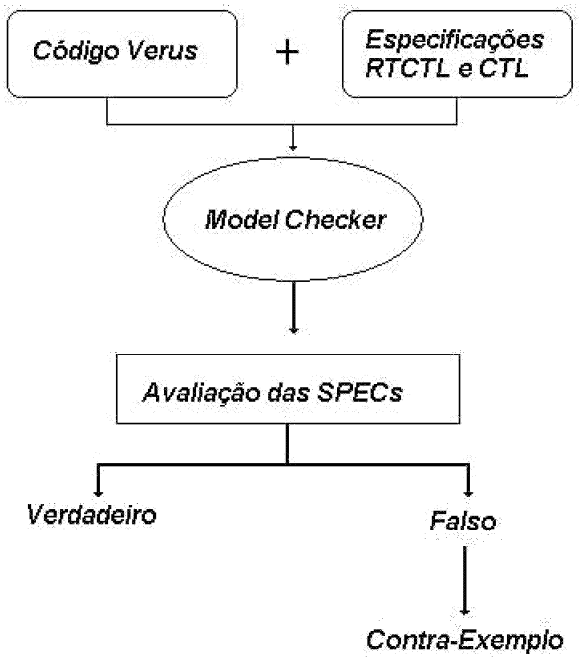

Figura 4.2: Visão Geral do Processo de Modelagem em Verus

O *Verus* também dispõe de um mecanismo útil para a verificação do modelo, que é o mecanismo de contra-exemplos. Um contra-exemplo é uma seqüência de estados, onde é possível verificar os valores das variáveis em cada estado. Através dos estados de de suas transições no modelo, é possível verificar, por meio do contra-exemplo, o estado ou o caminho entre estados que não satisfazem a propriedade verificada.

Desta forma, *Verus* caracteriza-se como uma ferramenta baseada em *Symbolic Model Checking*, utilizando algoritmos baseados em BDDs para a avaliação de fórmulas da lógica temporal. Sérgio Campos e Edmundo Clarke apresentam o algoritmo para verificação de modelos baseada em *Symbolic Model Checking* utilizando BDDs, com uma extensão para a manipulação de propriedades temporais utilizando o operador *until* limitado [CC94].

4.1.2 Exemplo de modelo em Verus: Produtor e Consumidor

Um exemplo de utilização do *Verus* pode ser visto a seguir, através da modelagem do problema clássico de produtor e consumidor [Cam97]. Neste modelo, são utilizada quatro

variáveis:

- p : utilizada para contabilizar a quantidade de bens produzidos.
- c : utilizada para contabilizar a quantidade de bens consumidos.
- produce : utilizada para indicar que um bem é produzido nesta unidade de tempo.
- consume : utilizada para indicar que um bem é consumido nesta unidade de tempo.

```
int p, c;
boolean produce, consume;

producer()
{
  p = 0;
  produce = false;
  while (true) {
    wait(1);
    wait(1);
    wait(1);
    produce = true;
    p = p + 1;
    /* p = select{p, p + 1}; */
    wait(1);
    produce = false;
  };
}
```

Programa 4.1: Exemplo de código em *Verus*: Produtor

O código 4.1 modela o comportamento do processo de produção. Inicialmente, as variáveis *p* e *produce* são iniciadas com os valores zero e *falso*, respectivamente. Em seguida, temos um laço incondicional aonde é modelado o processo. Após três unidades de tempo, um bem é produzido, o que é indicado no modelo através das variáveis *produce*, que recebe o valor *verdadeiro* e *p*, que é incrementada em uma unidade. Na unidade de tempo seguinte volta-se a sinalizar a variável *produce* com o valor *falso*, indicando que naquele instante o bem não é mais produzido, e o processo é reiniciado através do laço.

```
consumer()
{
  c = 0;
  consume = false;
  while (true) {
    wait(1);
    if (p != c) {
      wait(1);
      consume = true;
      c = c + 1;
      wait(1);
      consume = false;
    };
  };
}
```

Programa 4.2: Exemplo de código em *Verus*: Consumidor

No código 4.2, é modelado o processo de consumo dos bens produzidos. Inicialmente, as variáveis *c* e *consume* recebem os valores zero e *falso*, respectivamente. Em seguida é realizado o processo, através da execução de comandos em um laço incondicional. A cada unidade de tempo é verificado se o número de bens produzidos é diferente do número de bens consumidos. Caso isto ocorra, o bem é então consumido. Isto é sinalizado no processo através do incremento da variável *c* e da atribuição do valor *verdadeiro* para a variável *consume*. No instante seguinte a variável *consume* volta a receber o valor *falso*, indicando que naquele instante nenhum bem está sendo consumido.

Como podemos observar, um bem é produzido em três unidades de tempo, enquanto é consumido em apenas uma, de forma que em nenhum momento haverá sobrecarga de bens produzidos.

As propriedades verificadas e os resultados da modelagem podem ser visualizadas no trecho de código 4.3

```
*** VERUS ***

Using BDD library version 1.0.
........
Compiled function producer.
```

```
......
Compiled function consumer.


Time to construct the model: User :     0.06 s  System   :     0.01 s

Spec.  is true    : AG EF(produce);
Spec.  is true    : AG(produce -> AF consume);
Result is       1 : MIN(produce, consume);
Result is       1 : MAX(produce, consume);
Spec.  is true    : AG (produce -> (p != c));


Execution information:

Time              - User               :     0.08 s  System   :     0.01 s
BDD nodes used    - Transition relation:        614  Total    :       2814
Bytes allocated   -                          424148
Boolean variables -                              14
States            - Total              :       1792  Reachable:         33
```

Programa 4.3: Resultado das especificações após verificação do modelo

A propriedade *AG EF(produce)* é utilizada para verificar se bens sempre podem ser produzidos.

A propriedade *AG (produce → AF consume)* é utilizada para verificar se um bem produzido será sempre consumido em algum estado futuro.

São verificadas duas propriedades para saber o tempo mínimo e máximo que um bem leva entre sua produção e seu consumo.

Por fim, a propriedade *AG (produce → (p != c)* é utilizada apenas para garantir que não haverá sobrecarga no sistema, ou seja, um bem produzido será consumido somente em um momento futuro.

4.2 SMV

O SMV é uma ferramenta de verificação formal de sistemas baseada em *Symbolic Model Checking*, adequada para verificação de projetos de *hardware*.

A especificação de um modelo em SMV abrange uma coleção de propriedades. Uma propriedade pode ser tão simples quanto uma instrução que verifique se um par de sinais nunca é tomado em um dado momento, ou abranger relações mais complexas entre os valores obtidos entre tempos de geração de sinais (lógica temporal).

Possui grande eficiência na verificação automática de propriedades de lógica combinatória e máquinas de estados finitos interativas. É capaz de produzir contra-exemplos, muito úteis na verificação de propriedades de lógicas de controle complexas.

Para grandes projetos, especialmente os que incluem componentes de *data path* substanciais, o usuário pode dividir a prova de corretude em pedaços menores para que o SMV realize a verificação. Para isto, há dois mecanismos: O método proposicional, onde verifica-se propriedades de lógica temporal de uma parte do sistema, utilizando esta verificação como suposições quando da verificação de outras partes do sistema, e o método de refinamento, onde utiliza-se um modelo de alto nível do sistema como uma especificação e verifica-se separadamente que cada componente do sistema implementa sua parte do alto nível da especificação.

Podemos considerar, com exemplo, a seguinte descrição de circuito combinatório muito simples, com algumas afirmações adicionadas. Este exemplo é escrito em linguagem SMV nativa.

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;

  ack1 := req1;
  ack2 := req2 & ~req1;

  mutex : assert ~(ack1 & ack2);
  serve : assert (req1 | req2) -> (ack1 | ack2);
  waste1 : assert ack1 -> req1;
  waste2 : assert ack2 -> req2;
}
```

Programa 4.4: Descrição de circuito combinatório através de SMV

No código 4.4, podemos visualizar os componentes básicos de um módulo SMV. O

módulo possui quatro parâmetros: *req1*, *req2*, *ack1* e *ack2*, onde os dois primeiros são para entrada de dados e os dois últimos para saída. O módulo contém:

- Declaração de Tipos: Neste caso, os quatro sinais são declarados com o tipo *lógico*.
- Determinação de Sinal: Fornece funções lógicas para as variáveis de saída em termos das variáveis de entrada.
- Declarações: São propriedades a serem provadas.

O programa modela um "árbitro" baseado em prioridade, que poderia ser implementado através de um circuito de duas portas. As instruções *assert* especificam um número de propriedades que seriam interessantes de serem provadas sobre o circuito. Por exemplo, a propriedade denominada *mutex* diz que ambas as variáveis *ack1* e *ack2* não são verdadeiras na mesma unidade de tempo. A propriedade *serve* diz que se *req1* ou *req2* for verdadeiro, então uma das duas variáveis de saída será verdadeira.

4.3 STeP

STeP ou Stanford Temporal Prover [BBC+00] também é uma ferramenta para verificação formal das propriedades temporais de um sistema reativo, que implementa regras e diagramas de verificação, provendo suporte automático à prova de verificação de condições e permitindo verificação de modelo sempre que possível.

As propriedades temporais são implementadas através de fórmulas LTL e, para provar estas propriedades temporais, são usados Regras de Verificação e Verificação de Modelo (*Model Checking*). Para facilitar a prova de propriedades temporais também existem mecanismos de prova automática de teoremas e geração invariante.

Podemos definir o STeP como um arcabouço de um conjunto de ferramentas de métodos de verificação baseados em uma linguagem de descrição de um sistema comum e de uma linguagem de especificação. Assim, um sistema pode ser analisado de diversas formas. Dependendo do sistema e das propriedades a serem verificadas, ferramentas diferentes podem ser aplicadas de forma mais apropriada:

Model Checking Se o sistema for de estados finitos, propriedades temporais arbitrárias podem ser automaticamente estabelecidas ou refutadas, utilizando estado explícito ou *symbolic model checking*.

Geração Invariante Para muitas provas de verificação dedutivas, invariantes do sistema de força crescente devem ser coletadas, onde invariantes anteriores são utilizadas para estabelecer as subseqüentes. Geração automática de invariantes é utilizada para estabelecer um conjunto inicial de variantes.

Regras de Verificação Para provar propriedades simples de segurança, pode-se usar regras dedutivas, com o usuário fornecendo afirmações intermediárias quando necessário. Invariantes previamente estabelecidas são utilizadas para provar as condições de verificação requeridas, que são geradas automaticamente pelo sistema.

Diagramas de Verificação Podem ser fornecidos pelo usuário como uma abstração do sistema que prova uma propriedade particular em questão. Condições de verificação justificam a corretude do diagrama, enquanto um algoritmo verifica se o diagrama, de fato, prova a propriedade em si.

Abstração Desde que a abstração seja de estado finito, ela poderá ser verificada no modelo. Invariantes disponíveis melhoram a qualidade da abstração, permitindo que mais propriedades seja provadas.

O STeP provê uma interface gráfica para o usuário, implementada em Java, na forma de três editores: Editor de Sessão, Editor de Prova e Editor de Diagrama de Verificação.

4.4 SPIN

Desenvolvido pelo grupo de métodos formais e verificação da Bell Labs no começo dos anos 80, SPIN [Hol97] é um sistema genérico de verificação que suporta o projeto e a verificação de sistemas de processos assíncronos.

Seu principal foco reside em verificar de forma eficiente interação de processos de software e não hardware. Neste contexto, interação de processos podem ser especificadas em

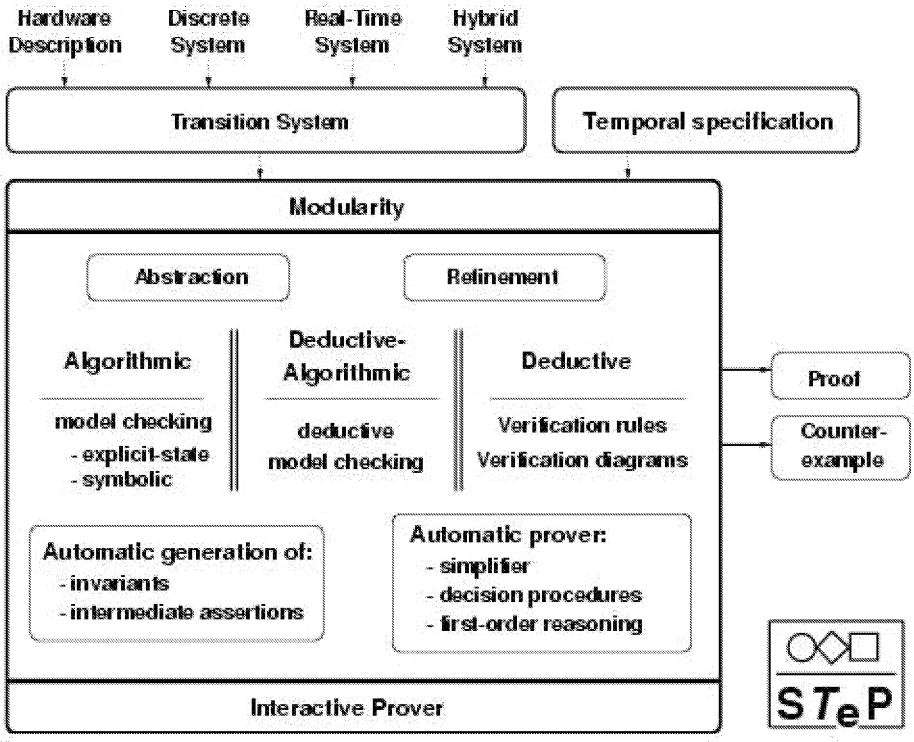

Figura 4.3: Visão Geral do STeP

SPIN com primitivas de reunião, com mensagens assíncronas passando através de canais de armazenamento, através de acessos a variáveis compartilhadas, ou com qualquer uma dessas combinações.

Para a construção do modelo, o SPIN utiliza uma linguagem de alto nível denominada PROMELA (PROcess MEta LAnguage), que caracteriza-se por ser não determinística, livremente baseada na notação da linguagem de comandos protegidos de Dijkstra e "toma por empréstimo"a notação para operações de entrada e saída da linguagem CSP de Hoare. Podemos ter uma visão do SPIN através do código 4.5 apresentado a seguir.

```
#define   NODES 3
#define   BUF_SIZE 1
chan input[NODES] = [BUF_SIZE] of {int};
chan broadcast = [0] of {int,int};
int leader_id[NODES];

proctype Node (int me; int myid) {
        int advert;
```

```
        leader_id[me] = myid;
        do
        :: input[me]?advert ->
                if
                :: advert < leader_id[me] ->
                        leader_id[me] = advert
                :: else -> skip
                fi
        :: true -> broadcast!me,leader_id[me]
        od
}
```

Programa 4.5: Exemplo de código PROMELA utilizado pelo SPIN

O sistema trabalha em modo "on-the-fly", ou seja, evita a necessidade de construir um grafo global de estados ou uma estrutura de Kripke como um pré-requisito para a verificação de algumas propriedades do sistema. Também suporta dinamicamente o crescimento e a contração da quantidade de processos, utilizando uma técnica de vetor "rubber state".

Propriedades de correção podem ser expressas utilizando-se LTL. Contudo, além de suportar todos os requerimentos apropriados expressáveis em lógica temporal linear, o SPIN também pode ser utilizado como um verificador "on-the-fly"eficiente para as propriedades de segurança mais básicas.

O SPIN pode ser utilizado de três maneiras básicas:

- Como um simulador, permitindo prototipação rápida com simulações aleatórias, guiadas ou interativas.
- Como um analisador exaustivo de espaço de estados, capaz de provar rigorosamente a validade da corretude de requisitos especificados pelo usuário (utilizando teoria de redução de ordem parcial para otimizar a pesquisa).
- Como um analisador de espaços de estados de bit, que pode validar até sistemas de protocolos muito grandes, com cobertura máxima do espaço de estados (uma técnica de prova de aproximação).

A ferramenta é escrita em linguagem C padrão ANSI, sendo portável através de todas as versões de sistemas operacionais UNIX. Também pode ser compilado para rodar em PCs com Linux, Windows95/98 ou Windows NT.

Capítulo 5

Fundamentos e Trabalhos Relacionados

5.1 Fundamentos Associados ao Trabalho

Tendo em vista os objetivos apresentados no capítulo, procurou-se concentrar a pesquisa em aplicações de verificação formal baseada em *symbolic model checking*, com a utilização de ferramentas voltadas a sistemas de tempo real.

A compreensão do funcionamento de autômatos finitos determinísticos também foi relevante para o entendimento da representação do autômato utilizado, e de como poder-se-ia construir um modelo baseado em lógica temporal a partir das informações iniciais obtidas. A característica de "explosão de estados"juntamente com a necessidade de se identificar possíveis desdobramentos do trabalho tornaram muito mais eficientes ainda a utilização de um *model checker*.

5.2 Trabalhos Relacionados

Técnicas de *Symbolic Model Checking* têm sido empregadas hoje especialmente para a verificação formal de projetos de hardware. Contudo, encontramos o emprego destas

técnicas em projetos industriais e verificação de software, também. Com o uso dessa técnica, é possível verificar sistemas de estados finitos com um grande número do espaço de estados (espaços com até 10^{30} estados podem ser pesquisados exaustivamente em minutos).

5.2.1 Exemplo de Verificação Formal em uma aplicação médica

No trabalho de Sérgio Campos, Edmund Clarke, W. Marrero e Marius Minea [CCMM95], são apresentados dois exemplos de verificação de modelos utilizando *Model Checking*. O primeiro, um exemplo realístico, descreve a verificação de um sistema de acompanhamento de pacientes, cujo modelo possui mais de 10^{13} estados possíveis, embora suas características temporais possam ser obtidas em alguns segundos.

O sistema, composto por nove processos - *filter*, *acquire*, *blood pressure*, *heart rate*, *temperature*, *alarm*, *display*, *record* e *audio* - reage a diversas situações anormais observadas em pacientes, sinalizando a ocorrência por meio de um alarme.

Os dados do paciente são lidos por meio de sensores de acompanhamento. O processo *acquire* realiza esta tarefa periodicamente, a cada 20 ms. Em seguida, os dados incorretos são eliminados por meio do processo *filter*. É feita, então, uma análise da condição clínica do paciente, através dos processos *blood pressure*, *heart rate* e *temperature*, que verificam, respectivamente, as condições da pressão sangüínea do paciente, sua freqüência cardíaca e temperatura. Conforme o resultado desta análise, um alarme sonoro pode ser disparado por meio dos processos *alarm* e *audio*. Por fim, os dados corretos coletados são enviados para uma tela e então são gravados em uma mídia, através dos processos *display* e *record*. O processo *filter* dura 3 ms enquanto os demais duram 2 ms em cada execução.

O objetivo da modelagem é verificar uma seqüência de execuções destes processos que tornem o sistema escalonável, utilizando tanto as restrições temporais de cada processo quanto uma ordem de prioridade para cada um deles. O algoritmo descrito no trabalho é capaz de determinar a existência de um processo cuja execução demanda uma quantidade ilimitada de tempo, o que torna o sistema não escalonável. Inicialmente, a ordem de prioridade, da mais alta para a mais baixa é: *acquire*,*filter*,*blood pressure*,*heart rate*,*temperature*,*display*,*recorder*,*alarm* e *audio*.

Os resultados obtidos pelo algoritmo são mostrados na tabela 5.1.

Processo	**Período**	Tempos de execução			
		(1)		(2)	
		min	max	min	max
acquire	20	1	1	1	1
filter	-	4	4	3	3
blood pressure	-	6	∞	2	2
heart rate	-	6	∞	2	4
temperature	-	6	∞	2	6
display	-	6	12	2	8
recorder	-	8	14	4	10
alarm	-	12	∞	6	10
audio	-	14	∞	2	2

Tabela 5.1: Seqüência 1 de Processos do Sistema de Acompanhamento Médico

Os tempos em (1) são os mínimos e máximos entre o início de um *acquire* e o final da execução do processo, enquanto os tempos em (2) são os mínimos em máximos entre o início e fim de execução de cada processo.

Algumas informações úteis são obtidas através deste modelo. Por exemplo, o disparo de um alarme, que é uma função crítica, pode ser adiado por outras funções como gravação de dados. Uma maneira de evitar o problema é aumentar a prioridade do processo de alarme. Assim, teríamos a seguinte ordem de prioridade: *acquire*,*filter*,*alarm*, *blood pressure*,*heart rate*,*temperature*,*display*,*recorder* e *audio*. Assim, temos os respectivos tempos de execução dos processos na tabela 5.2.

Processo	**Período**	Tempos de execução			
		(1)		(2)	
		min	max	min	max
acquire	20	1	1	1	1
filter	-	4	4	3	3
blood pressure	-	6	∞	2	2
heart rate	-	6	∞	2	6
temperature	-	6	∞	2	10
display	-	6	18	2	14
recorder	-	8	20	4	16
alarm	-	8	∞	2	2
audio	-	14	∞	6	∞

Tabela 5.2: Seqüência 2 de Processos do Sistema de Acompanhamento Médico

Uma comparação entre as duas tabelas nos dá que o tempo máximo de execução de alguns processos foi aumentado, embora nenhum processo ou carga adicional tenha sido incluídos no sistema, o que ainda torna o sistema não escalonável. Recorrendo ao

contra-exemplo gerado pelo SMV, através da propriedade verificada de que o processo *audio* sempre finaliza sua execução, é mostrada a seguinte seqüência: *acquire,blood pressure, alarm,heart rate,alarm,temperature,alarm,display,recorder,acquire,filter,...*, onde verificamos que há três execuções do processo *alarm* para uma só execução do processo *acquire*, causando uma sobrecarga no sistema. Uma solução simples indicada para o problema seria diminuir a prioridade do processo *alarm*, permitindo no projeto que múltiplos alarmes sejam manuseados corretamente. Assim, temos a seguinte ordem de prioridade que torna o sistema escalonável: *acquire,filter,blood pressure,heart rate,temperature,alarm,display,recorder* e *audio*.

5.2.2 Verificação Formal para Controle de Aeronave

No mesmo trabalho [CCMM95], é apresentada a aplicação da ferramenta SMV para verificação formal de um sistema de controle de aeronaves militares . O sistema possui controle de navegação, radar, armas e mostradores. Cada parte compões um subsistema. Similarmente ao exemplo anterior apresentado, o objetivo também é verificar a escalabilidade do sistema, a partir dos tempos de resposta de cada processo e da verificação do cumprimento do limite de tempo de execução para cada processo.

O controlador está dividido em sistemas e de seus sub-sistemas, cada um responsável por uma tarefa específica para controlar um componente da aeronave. O controle envolve a navegação, controle de radar, armas e exibição dos dados.

A restrição de tempo para cada sub-sistema depende de fatores como: precisão necessária do sistema, característica das respostas humanas e necessidades de hardware. Cada sub-sistema é implementado através de processos concorrentes. Para reforçar as diferentes restrições de tempo dos processos, utiliza-se prioridade de sincronização. Também é garantida a previsibilidade de processos.

Deste modo, empregou-se o SMV para verificar a correção funcional do sistema, enquanto a correção da temporização foi feita por meio de algoritmos quantitativos. Foi implementado tanto o planejador preemptivo de tarefas quanto o não-preemptivo, a fim de analisar os efeitos da preempção no tempo de resposta dos processos.

A escalabilidade foi determinada calculando-se os tempos de resposta de cada processo

e verificando se cada processo atingiu seu limite de tempo.

Através da modelagem, o trabalho mostra que o conjunto de processos do sistema é escalonável se for utilizado planejador preemptivo de tarefas. Contudo, a preempção não tem um grande impacto sobre os tempos de resposta. Exceto pelos processos mais críticos, todos os outros processos mantêm sua escalabilidade se o planejador não-preemptivo for utilizado. Além disso, a não-preempção faz com que o processo de liberação de armas perca seu prazo de tempo de execução, mas por um tempo relativamente pequeno.

Se o planejador preemptivo fosse caro, a leve redução de utilização da CPU poderia tornar o sistema completamente escalonável sem a necessidade de se alterar o planejador de tarefas. Com estas informações, o projetista pode avaliar facilmente o impacto de várias alternativas para melhorar o desempenho, sem a necessidade de alterar a implementação do sistema.

5.2.3 Prova de Correção de um Algoritmo de Controle de um Sistema Raid Nível 5

Vaziri, Lynch e Wing [VLW98] realizaram um trabalho de verificação formal em um algoritmo de controle genérico para a arquitetura de discos Raid 5, a partir de uma prototipagem feita por Courtright e Gibson.

O sistema Raid é composto de uma estrutura de discos e de uma controladora para receber operações dos usuários dos discos e executar de forma correta as operações de leitura e escrita nos discos específicos da estrutura. Problemas podem surgir durante estas operações. À medida em que mais discos são adicionados à estrutura, a disponibilidade de dados e a segurança da estrutura pode tendem a cair. Falhas catastróficas são consideradas, com perda total do conteúdo de dados de um disco. A arquitetura Raid 5 foi projetada para tolerância a uma falha de disco, além de utilização de paridade para efetuar redundância de dados. Nesta arquitetura específica, os dados são intercalados em blocos, e os blocos de paridade são distribuídos entre todos os discos da estrutura, e computados através de operação binária XOR nos blocos cobertos pela paridade. Assume-se que há $n + 1$ discos na estrutura. A controladora recebe operações de escrita e leitura do ambiente de forma seqüencial.

Os algoritmos da estrutura de disco são representados por grafos acíclicos dirigidos ou DAGs (Directed Acyclic Graphs). Cada nó deste DAG representa uma leitura ou gravação em disco ou uma operação XOR. Todas as operações de baixo nível de um DAG simples referem-se a um único grupo de paridade e representado operações de leitura e escrita nos setores dos discos.

Desta forma, foi utilizado autômato de E/S para modelar separadamente os DAGs do Raid 5 e verificar o algoritmo, além da ferramenta de verificação de modelos SMV para mostrar que os DAGs preservam a consistência do sistema.

5.2.4 Utilização de Métodos Formais para Protocolos de Criptografia e Comunicação de Dados

Também têm-se realizados trabalhos envolvendo a aplicação de métodos formais na validação de protocolos de criptografia e de comunicação de dados.

Conforme Catherine Meadows [Mea94], o emprego de métodos formais em protocolos de criptografia tem se concentrado na análise de protocolos já existentes, embora fique claro que tais métodos podem ser bastante eficientes e baratos de se implementar tanto no projeto de novos protocolos quanto no re-projeto dos já existentes. Acredita-se que esta concentração aconteça em razão da utilização de métodos formais para protocolos não ser muito antiga.

A incorporação de métodos formais no projeto de protocolos pode ser feita de duas maneiras. A primeira consiste em se projetar o protocolo de forma que o mesmo seja mais facilmente analisado com métodos formais. A segunda abordagem, que pode ser utilizada junto com a anterior, é baseada em camadas, onde a camada superior é um modelo relativamente abstrato, e a camada seguinte é então provada como sendo uma implementação da camada superior, até chegar-se a uma especificação detalhada ou ao próprio código produzido do protocolo.

Em se tratando de verificação formal de protocolos de comunicação, vem se tornando cada vez mais necessário o emprego de ferramentas baseadas em lógica temporal para modelar o comportamento de ambientes de redes face a diferentes configurações efetuadas. Esta necessidade ficou ainda mais clara com o advento de redes ad hoc, ou seja, redes onde

há pontos ou nós móveis.

A verificação formal de protocolos para redes convencionais, assim como para outras aplicações, exigia a construção de modelos mais triviais, pois as regras de relacionamento entre as variáveis que determinavam os possíveis estados eram mais simples e bem definidas. A explosão do número de estados dos modelos era bem mais controlável e previsível. Atualmente, existem vários protocolos para comunicação de dados de redes sem fio, tanto já implantados quanto em desenvolvimento. O fato de a rede não possuir todos os nós fixos exigiram regras mais complexas. Isto tornou maior o espaço de estados do modelo, exigindo o emprego de técnicas formais eficientes.

Bhargavan, Obradovic e Gunter [BOG99] apresentam em seu artigo a utilização do *model checker* SPIN, junto com o provador de teoremas interativo HOL para a verificação formal dos protocolo RIP (Routing Information Protocol) e AODV(Ad-Hoc On-Demand Distance Vector Protocol). A conclusão do trabalho é que é possível a utilização de métodos formais na verificação de protocolos de rede com esforço e rapidez razoáveis, mostrando, desta forma, que estas técnicas podem suplementar a utilização de outros meios de prova de segurança, como prova manual, simulação e testes.

5.2.5 Verificação Formal Centrada em Dados

Métodos de verificação formal são úteis para aplicações centradas em dados, como por exemplo verificação de existência de caminhos (rotas), que podem ser úteis para verificar o funcionamento de protocolos de acordo com a arquitetura de redes específicas. Ainda neste sentido, ferramentas de verificação formal podem ser utilizadas para verificar caminhos, de modo a se obter custos mínimos entre nodos de um grafo, que podem representar pontos em uma rede, ou cidades em uma malha aeroviária, por exemplo.

Neste trabalho, a verificação formal é realizada em uma aplicação também centrada em dados, onde têm-se a representação de ondas eletrocardiográficas previamente catalogadas, e se deseja descobrir se, dado um conjunto de regras para a geração destas ondas, é possível obter outras ondas ainda não catalogadas, de maneira a garantir que em um ambiente de monitoração de pacientes, todas as possíveis situações envolvendo o estado de saúde destes pacientes podem ser previstas. À medida em que novas ondas são descobertas, o modelo

cresce, de modo a comportar as novas ondas, até que todas sejam devidamente geradas e inscritas no modelo.

Capítulo 6

Modelagem

O reconhecimento de dados produzidos durante o eletrocardiograma é feito através da utilização de dois autômatos que, na verdade, atuam de forma conjunta e coordenada na identificação das ondas e das seqüências de ondas geradas.

As ondas identificadas através do primeiro autômato formarão a seqüência que deverá ser identificada pelo segundo autômato, de forma que, em conjunto com outras informações relevantes a respeito dos dados coletados pelo eletrocardiograma, seja fornecido um diagnóstico a respeito da situação do paciente e a ação necessária seja tomada.

Na metodologia utilizada para a verificação do mecanismo de reconhecimento de ondas, os autômatos são modelados separadamente, de forma que o modelo represente as características individuais de cada mecanismo e, desta forma, possa fornecer informações que possibilitem a identificação de falhas e sua correção. Desta forma, os dois modelos utilizados para verificar os mecanismos serão apresentados a seguir.

6.1 Modelagem do Autômato de Reconhecimento de Ondas Eletrocardiográficas

6.1.1 Descrição

As ondas são formadas pelas mudanças de voltagem dos sinais elétricos (deflexões). Hervaldo Carvalho [Car01] utilizou uma escala de aproximação logarítmica de cinco níveis para representar ondas, de acordo com seus conhecimentos médicos, de forma que os pontos de deflexões mais importantes pudessem ser bem representados.

A escala compreende níveis que podem ser: -2, -1, 0, 1 ou 2, representando, respectivamente deflexões ocorridas abaixo de -10 mV, entre -10 e -1 mV, acima de -1 mV e abaixo de 1 mV, entre 1 e 10 mV e, finalmente, acima de 10 mV.

As deflexões nas ondas podem ser superiores ou inferiores, observadas de acordo com a trajetória do sinal obtida pelo exame. Uma deflexão positiva é observada quando um sinal passa a ter uma voltagem mais baixa do que a atual. De forma análoga, na deflexão negativa o sinal passa a ter uma voltagem maior do que a atual.

Estas deflexões observadas ao longo do tempo é que formam as ondas eletrocardiográficas. Sua quantidade, duração e os respectivos pontos onde ocorrem dão origem às diversas formas de ondas, que podem, então, ser classificadas em um dos cinco tipos de ondas eletrocardiográficas.

6.1.2 Morfologia das Ondas Eletrocardiográficas

Baseado em seus conhecimento médicos, Carvalho [Car01] observou que as possíveis ondas produzidas por um eletrocardiógrafo possuíam no máximo três deflexões superiores na escala de representação, e também no máximo três inferiores na mesma escala. Isto é possível porque em amostras de pacientes com situações anormais algumas ondas podem não estar segmentadas, ou seja, pode não ser observado o segmento ou intervalo separando-as. Com isto, é possível que uma leitura normal esperada de um complexo QRS seguido de um segmento ST e de uma onda T possa ser obtida sem o segmento ST. Assim, não haveria uma leitura correta do complexo QRS e da onda T, e sim seria lida uma onda com

uma morfologia diferente das ondas conhecidas. Esta onda poderia ser, possivelmente, interpretada como um complexo QRS com características diferentes. A idéia principal desta verificação é detectar as possíveis formas das ondas encontradas na leitura de um ECG.

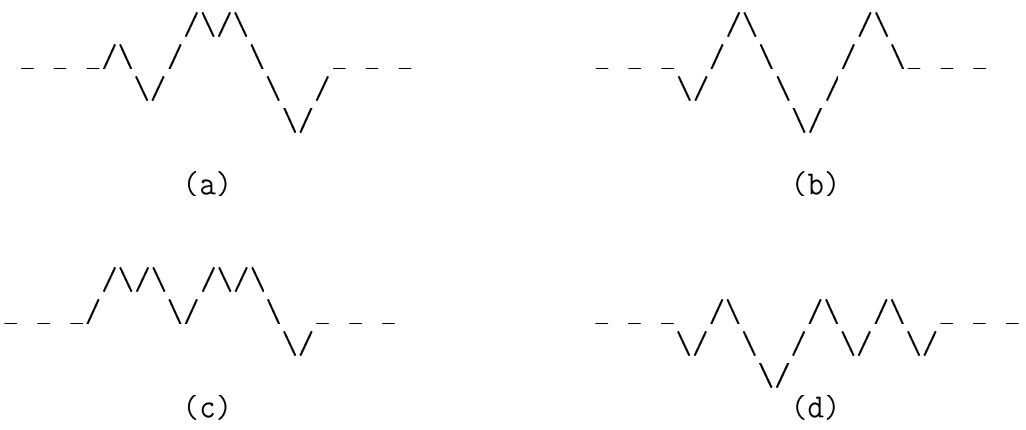

Figura 6.1: Ondas Válidas e Inválidas

Na figura 6.1, podemos observar que a onda (a) possui três deflexões superiores e três inferiores, constituindo-se como uma onda eletrocardiográfica válida. Igualmente, a onda (b) também caracteriza-se por ser uma onda eletrocardiográfica, uma vez que possui duas deflexões superiores e duas inferiores. Já a onda (c) possui quatro deflexões superiores e quatro inferiores, bem como a onda (d) possui três deflexões superiores, mas quatro inferiores, de forma que estas ondas não possuem as características observadas em ondas eletrocardiográficas [Car01].

Assim, corrigindo o autômato para o reconhecimento destas novas ondas, é possível então catalogá-las no sistema e, conforme a morfologia de cada uma delas, tratá-las da forma mais adequada. O importante é que qualquer onda possível de ser visualizada em uma amostra do ECG seja reconhecida pelo sistema.

A partir destas características morfológicas, foi modelado então o sistema de reconhecimento de ondas, implementado através de AFD. Para isto, o modelo foi estruturado com dois processos, de forma que o mecanismo de reconhecimento de ondas pudesse ser verificado. O primeiro tem por objetivo a geração de sinais elétricos nos níveis de escala de representação das ondas, de forma que todas as ondas possíveis sejam geradas. O segundo

processo faz a verificação do reconhecimento da onda gerada pelo autômato, prevendo sua correção por meio da inclusão de novos estados.

6.1.3 Processo de Geração de Ondas Eletrocardiográficas

Uma onda é caracterizada pela geração de sinais elétricos emitidos em intervalos de 1 ms (milisegundo). Embora esta emissão de sinais seja contínua, sua discretização é possível e bastante adequada, desde que não descaracterize a onda representada.

Desta forma, a representação sugerida no Life Guard discretiza os sinais gerados através de cinco níveis, conforme a intensidade elétrica de cada sinal. O reconhecimento de uma onda, utilizando-se AFD, prevê a ocorrência de sinais em forma seqüencial, até que um último sinal gerado leve a um estado final dentro do autômato.

Assim, foi possível modelar o processo de geração de sinais elétricos em unidades de tempo, de forma que uma seqüência de sinais emitidos caracterizasse a geração de uma onda, em conformidade com as ondas eletrocardiográficas a serem reconhecidas pelo sistema.

Geração de sinais elétricos

O processo de geração de sinais constitui-se basicamente em escolher qual o nível de transição, dentre os cinco utilizados na representação do sistema, que o sinal deve encontrar-se. Ou seja, sua função é gerar símbolos do alfabeto de entrada para a verificação do comportamento da máquina de estados ante o símbolo gerado.

A geração dos símbolos de entrada obedece às regras de morfologia das ondas observadas, de maneira que, ao final de todo o processo, a geração destes símbolos caracteriza a geração de uma onda eletrocardiográfica.

Devido ao *Verus* não trabalhar com números negativos, os níveis da escala tiveram que ser remapeados no modelo. Assim, em vez de um subconjunto de números inteiros (-2,-1,0,1,3), o modelo utilizou os números naturais (1,2,3,4,5), representando respectivamente a escala original.

Em um primeiro instante, as variáveis são iniciadas: *reconhecimento* e *geracao* com o valor *falso*, *inicio* com o valor *verdadeiro*, indicando o início da geração da onda, *nivel* com o valor inteiro 3, indicando o início da onda a partir do nível 0 da escala, e *transicoes_cima*,

transicoes_baixo e *estado* com o valor inteiro 0, indicando a quantidade de deflexões, no caso das duas primeiras variáveis, e o estado inicial do autômato, na terceira variável. É feito então o salto de uma unidade de tempo e a variável *inicio* volta a receber o valor lógico *falso*, de forma que somente no estado do modelo o início da geração de uma onda seja sinalizado, conforme podemos observar no trecho de código 6.1.

```
gerador()
{
  reconhecimento = false;
  geracao = false;
  inicio = true;
  nivel = 3;
  transicoes_cima = 0;
  transicoes_baixo = 0;
  estado = 0;
  wait(1);
  inicio = false;
```

Programa 6.1: Iniciação das variáveis do processo gerador de ondas eletrocardiográficas

O restante seguinte do processo encontra-se inserido em um laço incondicional (*while(true)*). Isto garante que todos os estados possíveis sejam gerados quando da montagem do modelo, ou seja, que todas as ondas possíveis serão geradas.

É feita, então, a escolha do próximo sinal a ser gerado. Caso o sinal atual seja 3 (nível 0), o estado seguinte é escolhido de forma não determinística, através da instrução *select*, entre os sinais 1,2,4,5 (-2,-1,1,2, respectivamente). Caso o novo estado escolhido seja menor que o atual, uma variável lógica chamada *nivel_anterior_maior*, utilizada para a detecção de deflexões, recebe o valor *verdadeiro*, caso contrário, recebe o valor *falso*, conforme podemos observar no trecho de código 6.2.

```
if (nivel == 3) {
        nivel = select {1,2,4,5};
        if (nivel < 3)
                nivel_anterior_maior = true;
        else
                nivel_anterior_maior = false;
}
```

Programa 6.2: Escolha não determinística dos níveis do sinal elétrico a partir do nível 0

Se o sinal atual for 4 (nível 1), é verificada então o número de deflexões, a fim de determinar qual o próximo sinal a ser gerado. Caso haja menos de três deflexões em ambos os sentidos, qualquer um dos outros sinais poderá ser gerado de forma não determinística. Se houver, porém, apenas menos de três deflexões superiores, o novo sinal a ser gerado é escolhido entre os níveis 0, indicando o final de uma onda, ou 5, que está acima do atual, levando a mais uma deflexão superior. Caso haja menos de três deflexões inferiores, o próximo sinal deverá ser gerado ou no nível 0, implicando na geração da onda, ou 1 ou 2, níveis inferiores ao atual, indicando a futura ocorrência de mais uma deflexão inferior, conforme observamos no trecho de código 6.3.

```
if (nivel == 4) {
        if (transicoes_cima < 3 && transicoes_baixo < 3)
                nivel = select {1,2,3,5};
        else
                if (transicoes_cima < 3)
                        nivel = select {3,5};
                else
                        if (transicoes_baixo < 3)
                                nivel = select {1,2,3};
                        else
                                nivel = 3;
...
```

Programa 6.3: Escolha não determinística dos níveis do sinal elétrico a partir do nível 1

Em seguida, é verificado se o novo sinal gerado encontra-se em um nível inferior ao atual. Caso isso ocorra, verifica-se se o valor da variável lógica *nivel_anterior_maior* é falso, indicando que, em relação ao estado atual 4, temos um nível anterior maior e um nível seguinte maior, o que implica em uma deflexão. Desta forma, a variável *transicoes_cima* é incrementada em uma unidade, para que a variável *nivel_anterior_maior* recebe o valor *verdadeiro*, já indicando que o nível atual 1 é maior que o novo. De forma análoga, esta comparação é feita quando o sinal gerado está no nível superior ao atual. Caso o valor da variável *nivel_anterior_maior* seja *verdadeiro*, temos então uma deflexão, o que implica em adicionar de uma unidade o valor da variável *transicoes_baixo* e sinalizar a variável lógica *nivel_anterior_maior* com o valor *falso*, indicando que o nível atual é menor do que o seguinte.

```
if (nivel < 4)  {
        if (!nivel_anterior_maior)
                transicoes_cima = transicoes_cima + 1;
        nivel_anterior_maior = true;
}
else   {
        if (nivel_anterior_maior)
                transicoes_baixo = transicoes_baixo + 1;
        nivel_anterior_maior = false;
}
...
```

Programa 6.4: Contabilização do número de deflexões a partir do nível 1

Por fim, testa-se se o sinal foi gerado no nível 0 (3, no modelo). Caso isto ocorra, é indicada então a finalização de geração de uma onda por meio da variável lógica *geracao*, que recebe o valor *verdadeiro*, conforme observado no trecho de código 6.5.

```
if (nivel == 3)
        geracao = true;
...
```

Programa 6.5: Determinação de geração de uma onda

Assim, a geração de uma onda é verificada ou quando o sinal gerado de forma não determinística situa-se no nível 0 da escala ou quando o número máximo de deflexões positivas ou negativas é atingido.

A mesma lógica é utilizada para o tratamento da geração de um sinal no nivel 4 (sinal 2). No caso de sinais gerados em nívels extremos (-1 ou 2), quaisquer outros níveis da escala poderão ser tomados de forma não determinística, e o valor da variável inteira *transicoes_cima* ou *transicoes_baixo* é sempre incrementado de uma unidade, conforme o nível do sinal gerado. Da mesma forma, a variável *nivel_anterior_maior* sempre tem o valor *verdadeiro*, caso o nível gerado seja 2, ou *falso*, caso o nível gerado seja -1.

Em seguida é feito o salto de uma unidade de tempo, a fim de que as mudanças efetuadas tenham efeito na configuração do estado atual do modelo. Em seguida, é verificado se houve a geração de uma onda, por meio da variável lógica *geracao*, que tem o seu valor inicial alterado de *falso* para *verdadeiro* sempre que o sinal atingir o nível 0, conforme o trecho de

código 6.5 . Se isto ocorrer, o processo é então preparado para a geração de outra onda, de forma que, neste instante, os valores das variáveis lógicas *geracao* e *reconhecimento* recebem o valor *falso*, e a variável lógica *inicio* recebe o valor *verdadeiro*, sinalizando o início da geração. As variáveis inteiras *nivel*, *transicoes_baixo* e *transicoes_cima* são iniciadas com o valor 0, para que, já em outra unidade de tempo, ocorra o reinício da geração da nova onda, por meio da atribuição de valores às variáveis *inicio* para *falso*, *estado* para 0, levando ao autômato a seu estado inicial, e *nivel* para 3, indicando o primeiro sinal gerado no nível 0 da escala, conforme descrito no trecho de código 6.6.

```
wait(1);
if (geracao)  {
        geracao = false;
        inicio = true;
        reconhecimento = false;
        nivel = 0;
        transicoes_cima = 0;
        transicoes_baixo = 0;
        wait(1);
        inicio = false;
        estado = 0;
        nivel = 3;
        wait(1);
}
```

Programa 6.6: Geração de uma onda e reinício do processo de geração

6.1.4 Processo para Verificação do Reconhecimento de Ondas

O reconhecimento das ondas produzidas pelo processo anterior ocorre neste processo, cuja codificação é feita a partir da tabela de transições de estado do AFD do Life Guard. Seu objetivo consiste em, dado estado atual do autômato e o valor do símbolo do alfabeto de entrada, definir qual o próximo estado a ser tomado. Além disto, caso o símbolo gerado leve a um estado não pertencente ao conjunto de estados do autômato, têm-se uma situação onde uma onda com as mesmas propriedades de ondas eletrocardiográficas é gerada, sem contudo ser reconhecida pelo autômato.

O processo utiliza duas variáveis principais para encontrar o próximo estado na tabela: *estado*, que armazena o atual estado do autômato, e *nivel*, alterada apenas no processo

de geração de sinais, contendo o atual nível dentro os cinco da escala onde encontra-se o sinal elétrico. Assim, é verificado qual o estado atual e qual o nível de escala do sinal, a fim de que seja escolhido o estado seguinte, percorrendo-se, assim, o autômato, até que um estado final seja tomado. Neste instante, uma variável lógica chamada *reconhecimento* é ativada com o valor lógico *verdadeiro*, indicando que a onda gerada pelo processo anterior foi reconhecida no autômato. Além disto, o processo também encarrega-se de ativar o valor 0 para a variável *estado*, a fim de retornar ao valor inicial do autômato para a geração de uma nova onda.

Conforme podemos observar na tabela de transições de estado (tabela 3.1), para determinados pares de estados e entradas do alfabeto, a função de transição não possui valores no contra-domínio. Por exemplo, para o par de entradas 84 e 2, $f(84, 2) = \emptyset$. Neste caso, para fins de modelagem, foi criado o estado imaginário 254, de forma que em todas as situações em que $f(E, V) = \emptyset$, têm-se no modelo $f(E, V) = 254$.

Este processo modela, na verdade, o caminhamento que se faz através da máquina de estados para reconhecimento de ondas, conforme o símbolo do alfabeto de entrada gerado no processo anterior. Ou seja, o que se modela neste processo é a função de transição $f(E, V)$ do autômato. Um erro no autômato é detectado quando é gerado quando a função de transição $f(E, V)$ leva ao conjunto vazio, ou seja, quando dado um símbolo de entrada e um estado atual do autômato, não existe um estado seguinte definido. Neste caso, a correção deve ser feita acrescentando o(s) estado(s) necessário(s) e repetindo-se a operação, até que pra todas as combinações possíveis de símbolos de entrada, $f(E, V) \neq \emptyset$.

Todo o corpo do processo está incluído em um laço incondicional (*while(true)*), uma vez que o processo de geração de ondas é c ontínuo, exigindo que o processo de reconhecimento também o seja.

O caminhamento na máquina de estados é feito por meio de instruções *if..then..else*. A cada sinal gerado pelo processo anterior, é verificado se, a partir do estado atual do autômato, qual o estado seguinte, conforme o valor do sinal gerado. Aqui, existem três situações possíveis:

1. O novo estado escolhido está presente no processo de reconhecimento e, conseqüentemente, na tabela de transição de estados. Neste caso, espera-se que um novo sinal seja gerado para que o processo possa se repetir.

2. O estado tomado é um estado final do autômato. Neste caso, a variável lógica *reconhecimento* passa a ter o valor *verdadeiro* e é feito então um salto de uma unidade de tempo. Em seguida, é sinalizado a volta do autômato ao seu estado inicial, a fim de que uma nova onda possa ser então gerada.

3. O valor da função de transição para o estado atual e o nível atual do sinal gerado é vazio, ou seja, não está prevista a ocorrência da onda gerada no autômato. Neste caso, o valor da variável *estado* passa a ser 254 (um estado inexistente no autômato), permanecendo desta forma até que a onda seja gerada por completo, de forma que o autômato volte ao estado inicial. Assim, é possível detectar em que ponto da geração da onda houve a falha no reconhecimento feito pelo autômato.

Há outra situação, onde a função de transição leva ao mesmo estado, conforme o símbolo do alfabeto de entrada gerado. Nestes casos, o que diferencia as ondas é a quantidade de vezes que isso deverá ocorrer, ou seja, a duração de cada onda.

Podemos observar no trecho de código 6.7 o tratamento no processo para os estados 0 (estado inicial), 254 (representa o conjunto vazio na tabela de transições) e 81.

```
automato()
{
   while(true) {

        if (estado == 0) {
                if (nivel == 3) estado = 0;
                if (nivel == 4) estado = 81;
                if (nivel == 5) estado = 131;
                if (nivel == 2) estado = 174;
                if (nivel == 1) estado = 218;
        }
        else
        if (estado == 254)   {     /* estado nao alcancado pelo automato */
                if (nivel == 3) {
                        wait(1);
                        estado = 0;
                }
        }
        else
        if (estado == 81) {
                if (nivel == 3) {
                        estado = 1;
```

```
                        reconhecimento = true;
                        wait(1);
                        estado = 0;
                }
                if (nivel == 4) estado = 81;
                if (nivel == 5) estado = 131;
                if (nivel == 2) estado = 161;
                if (nivel == 1) estado = 211;
        }
...
```

Programa 6.7: Processo de reconhecimento de ondas geradas

São verificadas três propriedades neste modelo:

AG EF(inicio) Este propriedade verifica se ondas sempre podem ser geradas, pois a cada geração de uma onda, a variável lógica *inicio* deverá ser receber o valor lógico *verdadeiro*, recebendo valores *falso* até o final de geração da onda.

AG (inicio → AF geracao) Esta propriedade verifica se sempre que num estado for observado o início da geração de uma onda, em algum estado futuro esta onda deverá ser gerada, ou seja, o valor lógico da variável *geracao* deverá ser verdadeiro ao final da geração de uma onda. Com isto, o valor lógico *verdadeiro* para esta propriedade garante a correta geração dos símbolos de entrada do autômato, conforme as características morfológicas das ondas.

AG (geracao → reconhecimento) Esta propriedade verifica que, se em um mesmo instante em que uma onda é gerada, ela também é reconhecida pela máquina de estados. Caso a função de transição leve a um conjunto vazio, representado no modelo como o inteiro 254, nunca será possível chegar a um estado final pertencente ao conjunto F do autômato. Isto levaria esta propriedade a receber o valor lógico *falso*. Utilizando-se o mecanismo de contra-exemplos do *Verus*, é possível saber em que ponto da geração de sinais a máquina de estados falhou, para então criar novos estados e assim corrigir a falha.

6.2 Modelagem do Mecanismo de Reconhecimento de Seqüências de Ondas Eletrocardiográficas

6.2.1 Descrição

A segunda máquina de estados utilizada pelo sistema tem por objetivo o reconhecimento da seqüência de ondas geradas em uma amostra de um eletrocardiograma. Existem diversas situações observáveis durante a leitura do eletrocardiógrafo que devem estar previstas no sistema, a fim de que o diagnóstico correto seja feito e as medidas necessárias sejam tomadas.

As seqüências obtidas em eletrocardiogramas de pacientes normais, conforme descrito no capítulo 3, envolvem a ocorrência de ondas P, QRS, T e, em raros casos, a onda U, nesta ordem. Contudo, fatores como ruídos no sinal e batimentos do coração em ritmos irregulares podem provocar seqüências anômalas, que podem indicar algum risco ao paciente. O sistema deve prever estas situações, assegurando assim alto grau de confiança e segurança na detecção de situações de risco ao paciente.

Nesta direção trabalhou-se em um modelo que pudesse descrever todas as possíveis seqüências obtidas em um exame de eletrocardiograma, a fim de gerar, com isto, uma máquina de estados capaz de prever estas seqüências e fornecer ao sistema um correto mecanismo de detecção de anomalias observadas através dos batimentos cardíacos.

Como este modelo aplica-se ao mecanismo de reconhecimento de seqüências sem considerar sua integração com o mecanismo de reconhecimento de ondas, o alfabeto de entrada utiliza os números 1 a 4 para representar, respectivamente, a ocorrência de ondas P, QRS, T e U dentro da seqüência. Assim, a máquina de estados utilizada neste modelo pode ser formalizada como: $M = (E, V, q, f, F)$, onde:

E é o conjunto de estados do autômato, formado pelos números inteiros compreendidos entre 0 e 127.

V é o conjunto de símbolos do alfabeto de entrada, formado pelos números naturais compreendidos entre 1 e 4.

q é o conjunto $\{q_0, q_1\}$ de estados iniciais do autômato, onde $q_0 = 0$ e $q_1 = 127$.

f é a função de transição de estados $f(E, V) = E$.

F é o conjunto de estados finais do autômato, onde $F \in E = \{0..39\}$

6.2.2 Processo Gerador de Seqüências de Ondas

Ao gerar as seqüências de ondas, devem ser observados os seguintes critérios:

- Uma seqüência pode começar ou com uma onda P ou com um complexo QRS.
- Caso uma onda P ou um complexo QRS seja repetido na seqüência, têm-se, então, o final da seqüência atual e o início de outra.
- Uma onda T só poderá ocorrer na seqüência se for precedida de um complexo QRS.

No primeiro instante, as variáveis de estado são iniciadas. As variáveis lógicas *geracao* e *reconhecimento* são iniciadas com o valor *falso*. A variável lógica *repete*, utilizada para indicar se uma onda P ou um complexo QRS já encontra-se presente na seqüência, também é iniciada com o valor *falso*. A variável lógica *inicio* é iniciada com o valor *verdadeiro*, indicando que neste momento o processo de geração de uma seqüência foi iniciado.

Neste mesmo instante, o estado inicial do autômato (0 ou 127) é escolhido de forma não determinística, através de uma instrução *select*. Esta escolha também implica na seleção do primeiro símbolo do alfabeto de entrada: caso o estado inicial escolhido seja 0, a primeira onda gerada é uma onda P, codificada através do inteiro 1. Caso o estado inicial escolhido seja 127, a onda tomada é então um complexo QRS, codificada pelo inteiro 2. Uma variável inteira *qtde_ondas*, utilizada para a contagem de ondas produzidas na seqüência é iniciada então com o valor 1. O processo é descrito no trecho de código 6.8.

```
sequencia()
{
  reconhecimento = false;
  geracao = false;
  comecoP = false;
  repete  = false;
  inicio = true;
  estado = select {0,127};
  if (estado == 0) {
```

```
        onda = 1;   /* onda P */
        comecoP = true;
}
else  onda = 2;   /* complexo QRS */
qtde_ondas = 1;
wait(1);
inicio = false;
```

Programa 6.8: Iniciação de variáveis no processo gerador de seqüências de ondas

No instante seguinte, são geradas as ondas restantes da seqüência. Esta parte do processo está inserida em um laço incondicional, codificado como *whiletrue*. Isto garante que todos os estados possíveis gerados a partir das restrições contidas no modelo e suas transições serão considerados no processo de montagem do mesmo, o que em outras palavras poder-se-ia traduzir como a garantia da geração de todas as seqüências possíveis de ondas eletrocardiográficas.

Neste instante, a variável lógica *inicio* recebe o valor *falso* e, já no corpo do laço do processo, uma nova onda será escolhida de forma não determinística. Aqui as restrições descritas anteriormente são implementadas. Caso a onda atual seja um complexo QRS (onda 2), a nova onda poderá ser qualquer uma das quatro: P(1), QRS(2), T(3) ou U(4). Caso a onda seja U(4), a onda seguinte poderá ser uma onda P(1) ou um complexo QRS(2). Por fim, caso a onda atual seja P(1) ou T(3), a onda seguinte poderá ser outra onda P(1), um complexo QRS(2) ou uma onda U(4).

Caso a onda escolhida seja uma onda P e a seqüência foi iniciada também com uma onda P, esta seqüência é então finalizada. Se a seqüência não foi iniciada com uma onda P, o valor da variável lógica *repete* é verificado e, caso ele seja verdadeiro, temos então uma repetição de ondas P, o que leva à finalização da seqüência. Por fim, se temos uma onda P, mas a seqüência iniciada por um complexo QRS e não há repetição da onda P, o valor da variável lógica *repete* passa a ser *verdadeiro*, a fim de indicar que outra ocorrência de uma onda P trata-se de uma repetição e a seqüência, por conseguinte, deverá ser então finalizada.

A mesma verificação é feita quando for gerado um complexo QRS. Caso a seqüência tenha sido iniciada também por um complexo QRS, ela é então finalizada. Se a seqüência tiver sido iniciada por uma onda P, é verificada então a repetição do complexo QRS dentro

da seqüência. Se houver repetição, a seqüência é finalizada. Caso contrário, a variável *repete* recebe o valor lógico *verdadeiro*, a fim de prevenir a repetição em outro estado do modelo.

Em quaisquer um dos casos descritos acima, a geração de uma seqüência é sinalizada no modelo por meio da variável lógica *geracao*, que deve receber o valor *verdadeiro*.

O processo pode ser visto no trecho de código 6.9

```
while(true)  {
        /* So pode haver onda T (3) se houver complexo QRS */
        if (onda == 2)   onda = select {1,2,3,4};
        else    if (onda == 4)  onda = select {1,2};
                else    onda = select {1,2,4};
        if (onda == 1)  {
                if (comecoP) geracao = true;
                else    {
                        if (repete)  geracao = true;
                        else  {
                                repete = true;
                                geracao = false;
                        }
                }
        }
        else    if (onda == 2)  {
                        if (comecoP)  {
                                if (repete) geracao = true;
                                else    {
                                        repete = true;
                                        geracao = false;
                                }
                        }
                        else    geracao = true;
                }

        qtde_ondas = qtde_ondas + 1;
...
```

Programa 6.9: Escolha da próxima onda da seqüência

Após a geração da onda e a verificação da finalização ou não da seqüência, é feito então o salto de uma unidade de tempo, através da instrução *wait(1)*. Em seguida, é verificado se houve a geração de uma seqüência de ondas. Se isto ocorrer, o modelo é então preparado

para gerar outra seqüência. Isto acontece da seguinte forma: As variáveis lógicas *geracao* e *reconhecimento* passam a receber o valor *falso*, enquanto a variável lógica *inicio* volta a receber o valor verdadeiro, indicando o início da nova seqüência. É feito o salto de uma unidade de tempo, e a primeira onda da seqüência é então escolhida, conforme o estado inicial determinado no processo de reconhecimento da seqüência, fazendo com que o processo seja então repetido, conforme pode-se verificar no trecho de código 6.10

```
wait(1);

if (geracao)  {
        geracao = false;
        inicio = true;
        reconhecimento = false;
        wait(1);
        inicio = false;
        qtde_ondas = 1;
        repete = false;
      comecoP = false;

        if (estado == 0) {
                onda = 1;  /* onda P */
                comecoP = true;
        }
        else    if (estado == 127)
                onda = 2;  /* complexo QRS */
        wait(1);
}
```

Programa 6.10: Geração de uma seqüência de ondas e início da geração da seqüência seguinte

6.2.3 Processo Verificador do Reconhecimento de Seqüências

Uma vez gerado o símbolo do alfabeto de entrada da máquina de estados no processo anterior, este processo encarrega-se de verificar o comportamento do autômato de acordo com este símbolo e com o estado atual deste autômato.

Todo o corpo do processo encontra-se inserido em um laço incondicional (*while(true)*), uma vez que a geração de seqüências também é feita incondicionalmente, até que todas as seqüências possíveis sejam geradas.

O tratamento da máquina de estados é todo feito por meio de instruções *if..then..else*. Inicialmente, é verificado o estado atual do autômato. Em seguida, é verificada então qual a onda gerada. Neste caso, temos então três possibilidades:

1. É escolhido um outro estado do autômato para que o processo se repita.

2. O autômato chega a um estado final. Neste caso, a variável lógica *reconhecimento* recebe o valor *verdadeiro* e é feito o salto de uma unidade de tempo. Em seguida, leva-se o autômato ao seu estado inicial, a fim de gerar outra seqüência de ondas.

3. Não há um estado posterior previsto no autômato para o símbolo do alfabeto de entrada gerado. Neste caso, o estado passa a receber o valor 125, não existente no autômato, a fim de indicar no contra-exemplo o ponto em que a máquina de estados falhou no reconhecimento da seqüência.

Sempre que estados novos necessitarem ser criados, este processo deverá ser alterado, uma vez que o mesmo reflete todo o comportamento da máquina de estados para o reconhecimento de seqüências de ondas.

O trecho de código 6.11 descreve o tratamento dos estados 0, 127 (estados iniciais para ondas P e QRS, respectivamente) e 40 (estado não inicial de uma onda P).

```
automato()
{
   while(true) {

        if (estado == 0) {              /*  onda P */
                if (onda == 1)
                        /* A sequencia ainda nao esta iniciada */
                        if (qtde_ondas == 1) estado = 0;
                        else {
                                estado = 3;       /* sequencia P - P */
                                reconhecimento = true;
                                wait(1);
                                estado = 0;
                        }
                if (onda == 2) estado = 60;       /* complexo QRS */
                if (onda == 3) estado = 125;      /* onda T */
                if (onda == 4) estado = 101;      /* onda U */
        }
        else
```

```
        if (estado == 127) {            /*  complexo QRS */
                if (onda == 1) estado = 40;        /* onda P */
                if (onda == 2)
                        /* A sequencia ainda nao esta iniciada */
                        if (qtde_ondas == 1)  estado = 127;
                        else  {
                                estado = 4;      /* sequencia QRS - QRS */
                                reconhecimento = true;
                                wait(1);
                                estado = 127;
                        }
                if (onda == 3) estado = 81;       /* onda T */
                if (onda == 4) estado = 102;      /* onda U */
        }
        else
        if (estado == 40) {            /*  onda P */
                if (onda == 1) {
                        estado = 9;       /* final de sequencia  */
                        reconhecimento = true;
                        wait(1);
                        estado = 0;
                }
                if (onda == 2) {
                        estado = 8;       /* final de sequencia  */
                        reconhecimento = true;
                        wait(1);
                        estado = 127;
                }
                if (onda == 3) estado = 125;      /* segmento PT */
                if (onda == 4) estado = 104;      /* segmento PU */
        }
...
```

Programa 6.11: Processo de reconhecimento de seqüências de ondas

Da mesma forma que o modelo do autômato anterior, neste modelo também são verificadas três propriedades importantes:

AG EF(inicio) Este propriedade verifica se seqüências sempre podem ser geradas, pois a cada geracao de uma seqüência, a variável lógica *inicio* deverá ser rotulada com o valor lógico *verdadeiro*, recebendo valores *falso* até o final de geração da seqüência.

AG (inicio → AF geracao) Esta propriedade verifica se sempre que num estado for ob-

servado o início da geração de uma seqüência, em algum estado futuro esta seqüência deverá ser gerada, ou seja, o valor lógico da variável *geracao* deverá ser verdadeiro ao final da geração de uma seqüência de ondas. Com isto, o valor lógico *verdadeiro* para esta propriedade garante a correta geração dos símbolos de entrada do autômato.

AG (geracao → reconhecimento) Esta propriedade verifica que, se em um mesmo instante em que uma seqüência é gerada, ela também é reconhecida pela máquina de estados. Caso a função de transição leve a um conjunto vazio, representado no modelo como o inteiro 125, nunca será possível chegar a um estado final pertencente ao conjunto F do autômato. Isto levaria esta propriedade a receber o valor lógico *falso*. Utilizando-se o mecanismo de contra-exemplos do *Verus*, é possível saber em que ponto da geração de sinais a máquina de estados falhou, para então criar novos estados e assim corrigir a falha.

6.3 Resultados Obtidos

6.3.1 Modelo de Verificação do Reconhecimento de Ondas Eletrocardiográficas

O modelo de verificação de reconhecimento de ondas é formado por dois processos: um para gerar todas as possíveis ondas eletrocardiográficas, de acordo com a representação em cinco níveis adotada para representar ondas reais, e outro para percorrer o autômato, de acordo com o estado atual e com o nível de sinal gerado.

Caso o modelo gere uma onda não prevista pelo autômato, a propriedade *AG(geracao → reconhecimento)* não deve ser satisfeita, ou seja tem o valor lógico *falso*, pois não será atingido nenhum estado final do autômato.

Utilizando o mecanismo de contra-exemplo do *Verus*, é possível saber exatamente em que ponto do autômato ocorreu a falha. Um símbolo de entrada não previsto para um estado do autômato levará o mesmo a tomar o estado seguinte como 254. Assim, no estado do modelo em que o estado do autômato passa a ter o valor 254, têm-se então o ponto da falha.

Desta forma, a partir do autômato inicial projetado por Carvalho [Car01], onde 59 ondas eletrocardiográficas eram previstas, foi possível aplicar a metodologia baseada em verificação de modelos e corrigir este autômato, de maneira que outras ondas pudessem ser reconhecidas. Com isso, foi feito o re-projeto do autômato, adicionando mais estados de sorte que as novas ondas encontradas fossem reconhecidas.

Um trecho de contra-exemplo gerado pelo *Verus* para o modelo de verificação de reconhecimento de ondas pode ser observado em 6.12.

```
*** VERUS ***

Using BDD library version 1.0.
read_ordering_file wave19.ord.
.......
Compiled function gerador.
.............................
Compiled function automato.


Time to construct the model: User :   112.88 s  System   :     0.13 s

Spec.  is true    : AG (inicio -> AF geracao);
No state satisfies the formula.
f & reachable no states.
Spec.  is false   : AG (geracao -> reconhecimento);
state   1:  inicio !geracao !reconhecimento estado=  0 nivel=  3
state   2: !inicio !geracao !reconhecimento estado=174 nivel=  2
state   3: !inicio !geracao !reconhecimento estado=100 nivel=  4
state   4: !inicio !geracao !reconhecimento estado=219 nivel=  1
state   5: !inicio !geracao !reconhecimento estado=177 nivel=  2
state   6: !inicio !geracao !reconhecimento estado=254 nivel=  4
state   7: !inicio  geracao !reconhecimento estado=254 nivel=  3

Execution information:

Time              - User               :   141.72 s  System   :     0.19 s
BDD nodes used    - Transition relation:      37244  Total    :     284519
Bytes allocated   -                       12660572
Boolean variables -                             70
States            - Total              : 1.59878e+17  Reachable:       1142
```

Programa 6.12: Contra-exemplo do modelo de verificação reconhecimento de ondas

No contra-exemplo, a falha ocorre durante a transição do estado 5 para o estado 6. O estado atual do autômato é 166, e o símbolo gerado é 4, uma situação não prevista no autômato. Desta forma, até a geração da onda, o modelo assume o valor 254 para o estado do autômato.

É possível identificar também que no passo 7 do contra-exemplo (state 7) ocorre a geração da onda. Desta forma, é possível corrigir o autômato, por meio da adição de novos estados que tornem possível o reconhecimento da onda gerada e identificada no contra-exemplo.

O autômato original previa até 80 estados finais, ou seja, 80 ondas. Para comportar sinais no nível 1 (4 no modelo), foram previstos inicialmente 50 estados, que poderiam variar do 81 ao 130. Para o nível 2 (5 no modelo), foram reservados 30 estados, variando entre 131 e 160. Para o nível -1 (2 no modelo), foi reservada a faixa de 161 a 210, totalizando 50 estados. Finalmente, para o nível -2 (1 no modelo), a faixa de 211 a 240 foi reservada, abrangendo 30 estados. Efetivamente, o autômato inicial utilizava 59 estados finais, 32 estados no nível 1 (81 ao 112), 19 estados no nível 2 (131 a 149), 27 estados no nível -1 (161 a 187) e 15 estados no nível -2 (211 a 225).

Desta forma, foram criados mais estados, de forma a comportar todas as ondas a serem geradas no modelo. Assim, o modelo prevê os seguintes estados:

- 0 : estado inicial do autômato.
- 300 a 399: São estados criados para comportar os sinais gerados no nível -2.
- 400 a 499: Criados para comportar os sinais gerados no nível -1.
- 1400 a 1454: Para comportar sinais gerados no nível -1.
- 500 a 799: Esta faixa foi utilizada para comportar os possíveis estados finais do autômato.
- 1500 a 1502: Estados finais do autômato.
- 800 a 899: Criados para comportar os sinais gerados no nível 1.
- 1800 a 1849: Criados para comportar sinais gerados no nível 1.

- 900 a 999: Foram criados para comportar sinais gerados no nível 2.

Assim, para corrigir o exemplo apresentado no código 6.12, foram utilizados os estados 802 e 701, de forma que $f(177, 1) = 802$ e $f(802, 0) = 701$.

Desta forma, a cada contra-exemplo gerado a partir da montagem do modelo, criava-se novos estados, ampliando os valores da função de transição do autômato. Com isto, foram geradas **303** novas ondas que, adicionadas às 59 reconhecidas pelo autômato, constituem o conjunto de todas as ondas que satisfazem às especificações descritas no capítulo 6.

Enquanto a propriedade AG *(geracao* $\rightarrow$ *reconhecimento)* recebesse o valor lógico *falso*, o contra-exemplo era obtido e os estados necessários eram criados, a fim de corrigir a falha no autômato. Por fim, após a obtenção de 303 contra-exemplos, a propriedade recebeu o valor *verdadeiro*, conforme mostrado no trecho de código 6.13.

```
*** VERUS ***

Using BDD library version 1.0.
read_ordering_file wave19.ord.
.......
Compiled function gerador.
............
Compiled function automato.


Time to construct the model: User :   114.85 s  System   :     0.27 s

Spec.  is true    : AG EF(inicio);
Spec.  is true    : AG (inicio -> AF geracao);
Spec.  is true    : AG (geracao -> reconhecimento);


Execution information:

Time              - User               :   116.57 s  System   :     0.27 s
BDD nodes used    - Transition relation:      37219  Total    :     283488
Bytes allocated   -                        10123476
Boolean variables -                              70
States            - Total              : 1.59878e+17  Reachable:       1130
```

Programa 6.13: Propriedades do Modelo após a obtenção de 303 contra-exemplos

Assim, a tabela de transição de estados foi estendida, de forma que o autômato possa reconhecer todas as possíveis ondas geradas. Inicialmente, foram identificadas 52 ondas que podem, potencialmente, ser observadas em um eletrocardiograma. Com isso, o sistema poderá tratar cada uma destas ondas, através do estado final na tabela de transições, de forma que elas possam ser classificadas em um dos quatro tipos diferentes de ondas (P, QRS, T ou U), e utilizadas como alfabeto de entrada para o segundo autômato, que faz o reconhecimento de seqüências de ondas geradas.

6.3.2 Modelo de Verificação do Reconhecimento de Seqüências de Ondas

De maneira similar ao modelo anterior, este modelo também utiliza dois processos para a verificação do reconhecimento de seqüências de ondas eletrocardiográficas.

Neste caso, contudo, a metodologia não foi aplicável somente na detecção de seqüências, mas também e, principalmente, no projeto de um autômato finito determinístico para o reconhecimento destas seqüências. Este autômato utiliza os seguintes estados:

- 0 : estado inicial do autômato, representando o início da seqüência por uma onda P.
- 127 : estado inicial do autômato, indicando o início da seqüência por um complexo QRS.
- 1 a 39 : estados finais do autômato.
- 40 a 59 : estados representando a ocorrência de ondas P.
- 60 a 79 : estados representando a ocorrência de complexos QRS.
- 80 a 99 : estados representando a ocorrência de ondas T.
- 100 a 119 : estados representando a ocorrência de ondas U.

No modelo foi utilizado o estado 125 para indicar um estado não previsto no autômato.

Inicialmente, as seqüências representadas no autômato foram as obtidas em eletrocardiogramas normais: P P-R QRS S-T T e P P-R QRS S-T T U-T U.

Assim, a cada montagem do modelo era verificada a propriedade *AG(geracao → reconhecimento)*. Caso seu valor fosse *falso*, era obtido o contra-exemplo, de maneira a determinar em que ponto do autômato a falha ocorreu, conforme podemos observar no trecho de código 6.14.

```
*** VERUS ***

Using BDD library version 1.0.
.......
Compiled function sequencia.
.............................................
Compiled function automato.


Time to construct the model: User :     2.26 s  System   :        0 s


Spec.  is true    : AG (inicio -> AF geracao);
No state satisfies the formula.
f & reachable no states.
Spec.  is false   : AG (geracao -> reconhecimento);
state 1:  inicio !geracao !reconhecimento comecoP !repete onda= 1 estado=  0
state 2: !inicio !geracao !reconhecimento comecoP  repete onda= 2 estado= 60
state 3: !inicio !geracao !reconhecimento comecoP  repete onda= 3 estado= 80
state 4: !inicio !geracao !reconhecimento comecoP  repete onda= 4 estado=100
state 5: !inicio  geracao !reconhecimento comecoP  repete onda= 2 estado=125

Execution information:

Time              - User               :     4.18 s  System   :        0 s
BDD nodes used    - Transition relation:      36618  Total    :     131085
Bytes allocated   -                          3806092
Boolean variables -                               45
States            - Total              : 1.55693e+10  Reachable:         91
```

Programa 6.14: Contra-exemplo obtido para o modelo de reconhecimento de seqüências

O exemplo acima reflete uma seqüência normal P P-R QRS S-T T U-T U seguida de uma seqüência com características diferentes da anterior, uma vez que já se inicia por um complexo QRS. Neste caso, o estado final 24 teve que ser criado, de forma que

$f(100, 2) = 24$.

Com isso, foram obtidas 38 diferentes seqüências, utilizando 38 estados finais, 4 estados para ondas P (40 a 43), 2 estados para complexos QRS (60 e 61), 3 para ondas T (80 a 82) e 11 estados para ondas U (100 a 110), gerando a nova tabela de transições de estados.

Neste autômato, contudo, todos os estados finais deverão ser considerados, uma vez que todas as seqüências geradas podem ocorrer potencialmente em um eletrocardiograma.

Após a obtenção dos 38 contra-exemplos, a propriedade *AG(geracao → reconhecimento)* recebeu o valor *verdadeiro*, conforme podemos observar no trecho de código 6.15.

```
*** VERUS ***

Using BDD library version 1.0.
.......
Compiled function sequencia.
.....................................................................
Compiled function automato.


Time to construct the model: User :     4.03 s  System   :     0.02 s

Spec.  is true    : AG EF(inicio);
Spec.  is true    : AG (inicio -> AF geracao);
Spec.  is true    : AG (geracao -> reconhecimento);


Execution information:

Time              - User               :     5.02 s  System   :     0.02 s
BDD nodes used    - Transition relation:      52726  Total    :     127169
Bytes allocated   -                        4240788
Boolean variables -                             45
States            - Total              : 2.36223e+10  Reachable:         83
```

Programa 6.15: Propriedades do Modelo após a obtenção de 38 contra-exemplos

Desta forma, o trabalho feito com esta modelagem resultou em um autômato de reconhecimento de seqüências de ondas, através da obtenção de sua tabela de transição de estados, a ser utilizada no mecanismo de reconhecimento de sinais do LIFE GUARD.

Capítulo 7

Conclusões e Trabalhos Futuros

7.1 Trabalhos Futuros

Deverá ser utilizado um autômato de onze níveis para representar com mais precisão a parte final de algumas ondas do complexo QRS, de forma que alguns estados do autômato atual de cinco níveis serão estados iniciais do novo autômato.

Com isso, o trabalho de verificação formal destes dois autômatos combinados será necessário, a fim de identificar as possíveis ondas não reconhecidas. Da mesma forma, se falhas forem encontradas, o modelo de verificação deve identificar o ponto em que elas ocorrem, bem como permitir que se faça a correção dos autômatos para identificar novas ondas.

A metodologia foi aplicada aos dois autômatos separadamente. Contudo, o funcionamento destes mecanismos no sistema de reconhecimento de dados eletrocardiográficos ocorre de forma integrada. As ondas reconhecidas através do primeiro autômato são classificadas conforme sua morfologia, tornando-se, então, símbolos do alfabeto de entrada do segundo autômato. Assim, a metodologia aplicada pode ser estendida, de forma a modelar o comportamento integrado deste mecanismo.

Outro aspecto a ser considerado diz respeito ao processo de geração de ondas proposto na metodologia. A geração dos sinais é seqüencial, de forma que apenas uma onda por vez é detectada e adicionada ao autômato de reconhecimento. Contudo, seria interessante

modificar o modelo para que fosse permitida a geração paralela de ondas, de forma que mais de um sinal fosse gerado no mesmo instante.

Seria também interessante considerar o uso de simuladores para gerar as ondas, de forma a comparar o mecanismo de geração da metodologia com os resultados obtidos nestes simuladores. Contudo, a metodologia prevê a geração do autômato de reconhecimento de ondas, para que o mesmo possa ser implementado no sistema de reconhecimento de sinais do LIFE GUARD, diferente de um simulador, onde apenas a geração de sinais é feita.

É possível e bastante adequado a utilização de técnicas de verificação formal para avaliar os diagnósticos emitidos pelo sistema, diante das possíveis situações observadas durante o acompanhamento do paciente. Isto complementaria o trabalho desenvolvido no reconhecimento de ondas e de seqüências, no que diz respeito à utilização de técnicas formais aplicadas na verificação do módulo de acompanhamento do paciente.

Um importante e potencial trabalho de verificação pode ser feito quando todos os módulos do sistema estiverem projetados, de forma que haja a integração entre eles, de forma que as possíveis falhas sejam detectadas e seus impactos sejam avaliados em relação ao grau de confiança do sistema.

7.2 Conclusão

Técnicas formais têm sido aplicadas com sucesso na verificação de projetos de circuitos lógicos e microprocessadores, como também na verificação de diversos sistemas embutidos aplicados em áreas como aviação, por exemplo. Os resultados obtidos neste trabalho mostram que técnicas de verificação formal podem ser muito bem aplicadas na verificação de aplicações biomédicas, uma vez que estas aplicações, em grande parte, são críticas e executadas em tempo real, exigindo um alto grau de confiança.

A metodologia aplicada para a verificação formal possibilitou que fossem encontradas falhas no mecanismo de reconhecimento de ondas, bem como permitiu que fosse feito o projeto de um autômato finito determinístico para o mecanismo de reconhecimento de seqüências de ondas, contribuindo de forma efetiva para o aumento do grau de confiança exigido por uma aplicação biomédica crítica de tempo real.

Desta forma, a metodologia proposta neste trabalho pode ser aplicada a outras apli-

cações não somente da área biomédica, como também de outras áreas que necessitem trabalhar com autômatos finitos determinísticos no reconhecimento de padrões para análise. Sua aplicação permite não somente a correção, como também auxilia no projeto dos autômatos, de maneira a garantir maior grau de confiança e precisão deste tipo de aplicação.

Bibliografia

[BBC+00] Nikolaj S. Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, 16(3):227–270, June 2000.

[BOG99] K. Bhargavan, D. Obradovic, and C. Gunter. Formal verification of standards for distance vector routing protocols, 1999.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Cam97] Sérgio Vale Aguiar Campos. Verus 0.9 - reference manual, March 1997.

[Car00] Hervaldo S. Carvalho. Lecture notes. 2000.

[Car01] Hervaldo Sampaio Carvalho. Contatos verbais, 2001.

[CBBL91] Tapan J. Chakraborty, Sudipta Bhawmik, Robert Bencivenga, and C. J. Lin. Enhanced controllability for i_{ddq} test sets using partial scan. *28th ACM/IEEE Design Automation Conference*, 1991.

[CC94] Sérgio Campos and Edmund M. Clarke. Real-time symbolic model checking for discrete time models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*. World Scientific Press, AMAST Series in Computing, May 1994.

[CC01] Sérgio Vale Aguiar Campos and Edmund M. Clarke. The verus language: representing time efficiently with BDDs. *Theoretical Computer Science*, 253(1):95–118, 2001.

[CCMM95] Sérgio Campos, Edmund Clarke, W. Marrero, and Marius Minea. Timing analysis of industrial real-time systems. In *Workshop on Industrial-strength Formal specification Techniques*, 1995.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. In *Conference record of the 10th ACM Symposium on Principles of Programming Languages (POPL)*, pages 117–126, 1983.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, third edition, 2001.

[dEdC] Laboratório de Eletrofisiologia de Curitiba. Ritmos do Coração. Disponível na url: http://www.lec.pucpr.br/body_rit.html.

[Flo] Xavier F. Flores. ECGs Interpretation. Disponível na url: http://www.utoledo.edu/~xflores/ekgs-interpretation.htm.

[Gar] Aitor Etxeberria Garin. Electrocardiograma. Disponível na url: http://es.geocities.com/simplex59/electrocardiograma.html.

[Gor00] Valentin Goranko. Temporal logics with reference pointers and computation tree logics. *Journal of Applied Non-Classical Logics*, 10(3-4), 2000.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[JG98] Dean Jenkins and Stephen Gerred. ECG Library. Disponível na url: http://www.ecglibrary.com/ecghome.html, 1998.

[MCC00] Autran Macêdo, Sérgio Campos, and Carlos R. V. Carvalho. Symbolic model checking: A new approach for solving scheduling problems. 2000.

[Mea94] Catherine A. Meadows. Formal verification of cryptographic protocols: A survey. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*. LNCS, Springer-Verlag, 1994.

[NMDB99] Maria G. V. Nunes, Alessandra Alaniz Macedo, Daniel Gomes Dosualdo, and Tatiana Barbosa. SCE123: Introdução à Compilação. Instituto de Ciências Matemáticas e Computação. Departamento de Computação e Estatística. Universidade Federal de São Paulo, 1999.

[SEM97] Sérgio Campos, Edmund M. Clarke, and Marius Minea. The verus tool: A quantitative approach to the formal verification of real-time systems. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 452–455. Springer Verlag, 1997.

[Som99] Fabio Somenzi. Binary decision diagrams. In *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[Sun98] Kim Sunesen. *Reasoning about Reactive Systems*. Ph.d. dissertation, University of Aarhus, Ny Munkegade, building 540, Department of Computer Science, December 1998.

[Vis98] Willem Visser. *Efficient CTL* Model Checking using Games and Automata*. Ph.d. thesis, Manchester University, June 1998.

[VLW98] Mandana Vaziri, Nancy A. Lynch, and Jeannette M. Wing. Proving correctness of a controller algorithm for the RAID level 5 system. In *Symposium on Fault-Tolerant Computing*, pages 16–25, 1998.

[Yan] Frank G. Yanowitz. ECG Outline. Disponível na url: http://www-medlib.med.utah.edu/kw/ecg/ecg_outline/.