Aplicações Práticas de Especificações Formais Orientadas a Objetos: Um Sistema para Processamento do Eletrocardiograma de Esforço

Gustavo H. M. B. Motta
Silvio R. L. Meira

Departamento de Informática, UFPE
e-mail: srlm@di.ufpe.br

Luis C. Carvalho

Núcleo de Estudos e Tecnologia em Engenharia Biomédica, UFPB

Sumário

Este trabalho descreve a experiência no uso de especificações formais orientadas a objetos na construção de um sistema para processamento automático do eletrocardiograma de esforço. A construção do sistema empregou uma metodologia de desenvolvimento de software orientado a objetos que tem como base MooZ, uma linguagem de especificação formal orientada a objetos, utilizada como notação para descrição, documentação e projeto do software. O uso de uma notação matemática abstrata, estruturada e bem definida deu o suporte para um bom entendimento do problema estudado ainda nos estágios iniciais do desenvolvimento.

Abstract

This work describes an experience on using object oriented formal specifications to implement an automatic exercise electrocardiograms processing system. The implementation followed an object oriented software development method, based on MooZ, an object oriented formal specification language, which was used as notation for software description, documentation and design. The use of an abstract, structured and well defined mathematical notation, allowed a good understanding of the in hand problem, even in the early stages of development.

1 Introdução

A motivação para realização deste trabalho foi experimentar a aplicação rigorosa de métodos matemáticos na construção de sistemas de software não-triviais, buscando evidenciar sua viabilidade como instrumento efetivo para o desenvolvimento de software. Isto para mostrar que a aplicação de tais métodos é não apenas viável, mas extremamente necessária e benéfica, mesmo com seus detratores afirmando que são difíceis de usar, pouco úteis, acadêmicos, de custo elevado e aplicáveis apenas a problemas triviais.

A Engenharia Biomédica (EB) foi a área de aplicação escolhida para experimentar especificações formais porque faz uso intensivo de software em sistemas que atuam em áreas críticas, como a medicina, mas que não emprega técnicas avançadas de Engenharia de Software (ES) para construção de seus programas. Nos atuais sistemas desenvolvidos na EB, o software desempenha um papel cada vez mais importante. A cada dia, torna-se mais complexo, exigindo grandes esforços para se construir e manutenir. Deve ser confiável, eficiente e facilmente reutilizável. Os sistemas mais avançados de hoje contam com mais de 20 processadores e 400.000 linhas de código C++ para realizar funções que componentes discretos faziam 13 anos atrás[10].

Entretanto, as técnicas empregadas para concebê-los são inadequadas para este propósito, sendo fortemente baseadas em considerações pragmáticas, não levando em consideração nenhum aspecto formal.

Neste trabalho, optou-se construir um sistema médico baseado em computador (SMBC) para processamento do eletrocardiograma (ECG) de esforço, visto que este apresenta características complexas, como processamento em tempo real dos sinais de ECG e técnicas sofisticadas para processamento de sinais. Resultados deste trabalho, divulgados em [18, 17], ressaltou para área de EB os benefícios e a importância do uso de um formalismo para desenvolvimento de software, particularmente para especificação e projeto.

A construção do sistema utilizou uma metodologia de desenvolvimento de software orientado a objetos, que tem como base MooZ[13], uma linguagem de especificação formal orientada a objetos, usada como notação para descrição, documentação e projeto de software. Um conjunto de testes preliminares, objetivando avaliar o sistema em funcionamento foi realizado com vinte indivíduos normais no Hospital Universitário da UFPB, com bons resultados.

2 Especificações Formais Orientadas a Objetos

2.1 Especificações Formais no Desenvolvimento de Software

Em Engenharia de Software, é sabido que nem sempre as implementações satisfazem o sistema proposto pela especificação dos seus requisitos. Os documentos de requisitos, em geral, definem as principais funções do sistema de forma adequada, mas muitos detalhes importantes para o projeto e a implementação não são precisamente descritos. Isto ocorre sempre que as descrições, mesmo as de boa qualidade, são feitas em linguagem natural, que exibem deficiências que as tornam inaceitáveis para um desenvolvimento rigoroso de software.

Estas descrições ou especificações informais são inerentemente ambíguas e inconsistentes e provocam erros de interpretação que são propagados ao longo do projeto e da implementação. Os erros, quase sempre, só são descobertos nos estágios finais do desenvolvimento do sistema, onde os custos para corrigí-los são muito altos.

Milner argumenta, em [16], que computação, como qualquer outra disciplina da engenharia, deve empregar teorias matemáticas a serviço do projeto; teorias estas que ajudem a fazer um bom projeto e a validar produtos. Afirma que tais teorias não precisam dizer nada sobre a natureza da computação; tudo o que elas devem fazer é ajudar na construção de sistemas de computação complexos e no gerenciamento desta complexidade.

A utilização de uma linguagem de especificação formal é a pedra fundamental para formalizar o processo de desenvolvimento de software. Linguagens de especificação possuem uma base teórica bem fundamentada, como a lógica de primeira ordem e teoria dos conjuntos ou álgebras polisortidas. A base matemática visa possibilitar a verificação de propriedades das especificações. Adicionalmente, é desejável que tenham mecanismos estruturais de decomposição e composição e grande expressividade, que facilitam a especificação e o gerenciamento de grandes sistemas. É importante que tenham um teoria de prova que permita verificar a equivalência semântica de especificações em diferentes graus de abstração, ou esquemas de transformação que permitam derivar especificações mais concretas a partir das mais abtratas.

2.2 Orientação a Objetos em Linguagens de Especificação Formal

O paradigma de orientação a objetos, desde a última década, introduziu nas linguagens de programação características fundamentais de encapsulação, abstração e modularização que resultaram em poderosos meios para definição e gerenciamento de sistemas grandes e complexos. Estes incluem os conceitos de objeto, classe e herança. Objetos são coleções de operações que compartilham um estado. Uma classe especifica um conjunto de objetos com estrutura e com-

PDF compression, OCR, web optimization

portamento comuns. Herança permite o resuso do comportamento e da estrutura de uma ou mais classes (superclasses) na definição de novas classes (subclasses)[22].

Nos últimos anos, muitas contribuições vêm sendo feitas para incorporação dos mecanismos de estruturação de orientação a objetos em linguagens de especificação formal, esperando alcançar os mesmos benefícios já obtidos com o paradigma em linguagens de programação. Linguagens de especificação formal mais tradicionais, como VDM-SL (Vienna Development Method - Specification Language)[9] e Z[19], não possuem uma notação poderosa o suficiente para concepção, estruturação e gerenciamento de grandes especificações.

Uma extensão de Z que suporta especificações modulares é apresentada em [15]. Inclui duas novas estruturas sintáticas, chamadas capítulo e documento. Um capítulo contém parte de uma especificação e inclui facilidades para compartilhamento de definições com outros capítulos, através de mecanismos de importação, exportação e parâmetros genéricos. Um documento agrupa capítulos que definem uma especificação.

Dentre as várias extensões realmente orientadas a objetos propostas para Z, destacam-se MooZ[13], Object-Z[6], Z^{++}[11] e OOZE[1]. Todas estas extensões introduzem os conceitos de objeto, classe e herança, embora sejam significativamente diferentes entre si. Um estudo comparativo que examina vários enfoques para usar ou estender a linguagem de especificação Z de uma maneira orientada a objetos é apresentado em [20].

MooZ, desenvolvida no Departamento de Informática da Universidade Federal de Pernambuco, foi a escolhida para especificação, projeto e documentação do SMBC para processamento do ECG de esforço, pela necessidade de experimentação da linguagem em desenvolvimento real de software.

3 O Problema

A experiência relatada neste trabalho foi a construção de um SMBC para processamento do eletrocardiograma de esforço com o emprego de MooZ para especificar, projetar e documentar o sistema. O problema a ser resolvido foi automatizar o teste de esforço ou teste ergométrico, uma técnica amplamente utilizada e universalmente aceita para diagnóstico de isquemia miocárdica.

Antes de dar prosseguimento à descrição do problema, vamos apresentar no quadro abaixo alguns conceitos de eletrocardiografia[7], que vão ajudar no entendimento dos requisitos do sistema.

> A eletrocardiografia é o estudo do registro gráfico das correntes elétricas originárias do músculo cardíaco, sendo valioso auxiliar no diagnóstico de numerosas doenças cardíacas. Este registro é chamado *eletrocardiograma* e permite a análise do funcionamento do coração do ponto de vista da sua atividade elétrica.
>
> A figura 1 ilustra um esquema de um ciclo cardíaco típico com os seus principais componentes. Uma seqüência de ciclos cardíacos representa uma seqüência de batimentos sucessivos do coração e forma um eletrocardiograma. Dentre os vários componentes que formam um ciclo cardiáco, destacam-se as ondas P,Q,R,S e T e os segmentos PR e ST.
>
> As forças elétricas do coração passíveis de serem registradas são resultantes de correntes extremamente variáveis. Esses potenciais se desenvolvem em três dimensões e sua adequada apreciação pode ser conseguida em relação a dois planos interseccionados. Para tanto, eletrodos exploradores devem ser colocados na periferia do corpo, constituíndo as chamadas *derivações eletrocardiográficas.*

O exame consiste numa prova realizada em esteiras ou bicicletas ergométricas, obedecendo a um protocolo que divide o exame em fases com tempo de duração e carga de trabalho específicos. O teste ergométrico exige monitorização eletrocardiográfica contínua e ao final de cada fase deve ser armazenado no computador de uma a doze derivações de ECG, juntamente com o valor da pressão arterial do paciente. A partir da análise das derivações do ECG armazenado para cada fase, são obtidas valiosas informações que são determinantes para o diagnóstico da isquemia miocárdica. O comportamento do segmento ST é seu elemento diagnóstico fundamental.

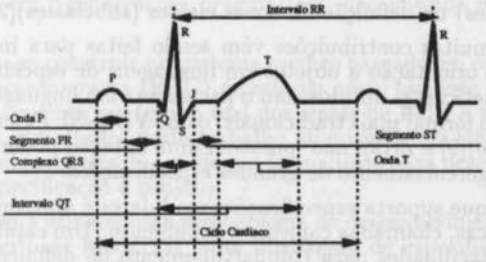

Figura 1: O ciclo cardíaco e seus principais constituintes

Seus desníveis, no decorrer do teste de esforço, constituem a alteração eletrocardiográfica que melhor correlação apresenta com os achados anatômicos da coronariografia, na insuficiência coronariana[12]. Os parâmetros mais importantes para o exame e que podem ser obtidos diretamente da análise do ECG são: duração do complexo QRS; integral do segmento ST; índice do segmento ST; amplitude das ondas Q,R,S e T; intervalo QT; amplitudes após 60 ms e 80 ms do fim do complexo QRS (ST-60 e ST-80) e freqüência cardíaca.

O exame é complexo e a análise do ECG dos diversos estágios é uma tarefa demorada e trabalhosa quando realizada manualmente. Sua automação, por outro lado, não é uma tarefa trivial, uma vez que envolve: conversão analógico-digital dos sinais das várias derivações de ECG durante o teste; visualização contínua em tempo real, no monitor de vídeo, das derivações amostradas; detecção em tempo real do complexo QRS; cálculo em tempo real da freqüência cardíaca; processamento para determinação de um ciclo cardíaco mediano de cada derivação registrada em cada estágio; determinação dos parâmetros de interesse para análise do ECG de esforço usando o ciclo cardíaco mediano de cada derivação dos diversos estágios e emissão de relatórios com as medidas calculadas.

4 O Desenvolvimento

4.1 Metodologia

A metodologia empregada no desenvolvimento deste sistema teve como principal característica o uso de MooZ. Por ser formal, MooZ permitiu uma maior segurança para que o projeto capturasse os requisitos críticos do sistema e que sua implementação fosse uma realização acurada do projeto. Por ser orientada a objetos, tem os poderes já consagrados do paradigma, assegurando uma visão coerente e unificada de todo o desenvolvimento.

4.1.1 MooZ

MooZ é uma linguagem que combina o poder de um formalismo já conhecido e bem estabelecido, a linguagem Z, com aqueles do paradigma de orientação a objetos. Uma especificação em MooZ consiste num conjunto de definições de classe que podem estar relacionadas por uma hierarquia. Uma destas classes tem o nome do sistema definido. O estado e as operações dos objetos desta classe especificam o sistema estudado. A forma geral de uma classe é mostrada na figura 2.

A cláusula givensets introduz uma lista de nomes de tipos indefinidos. Um tipo indefinido pode servir para o propósito de abstração ou generalização; um objeto do mundo real que não necessita de um modelo num determinado nível de abstração, pode ser representado por um tipo indefinido.

Class ⟨*Nome_da_Classe*⟩

givensets ⟨*lista_de_nomes_de_tipo*⟩

superclasses ⟨*lista_de_referencias_de_classe*⟩
⟨*definicoes_auxiliares*⟩

private ou public ⟨*lista_de_nomes_de_definicao*⟩

constants
⟨*lista_de_descricoes_axiomaticas*⟩
⟨*definicoes_auxiliares*⟩

state
⟨*esquema_anonimo*⟩ ou ⟨*restricao*⟩

initialstates
⟨*esquema*⟩
⟨*definicoes_auxiliares*⟩

operations
⟨*lista_de_definicoes*⟩

EndClass ⟨*Nome_da_Classe*⟩.

Figura 2: Forma geral de uma classe em MooZ

A cláusula **superclasses** introduz os nomes das classes herdadas. As definições que são visíveis numa classe, também o são em todas as suas subclasses. Colisão de nomes entre superclasses são resolvidas através de um mecanismo de troca de nomes. Toda definição visível numa classe – suas próprias definições e as de suas superclasses – corresponde a uma mensagem que pode ser enviada a um objeto desta classe. Esta visibilidade pode ser mudada usando as cláusulas **private** ou **public**. A cláusula **constants** introduz constantes globais.

Os componentes do estado de uma classe são aqueles das suas superclasses e os introduzidos pela cláusula **state**, onde o invariante da classe, uma possível restrição sobre os valores do estado, pode ser estabelecido. Os esquemas na cláusula **initialstates** objetivam definir o estado inicial de objetos da classe através de operações de iniciação. Portanto, se um esquema na cláusula **initialstates** define uma operação, os possíveis estados finais desta operação são os estados iniciais permitidos a objetos da classe. Mais de uma operação de iniciação podem ser definidas.

As operações de uma classe são aquelas das suas superclasses somando-se às introduzidas na cláusula **operations**. Operações definidas nas superclasses podem ser redefinidas nas subclasses. Quando uma operação O está definida em várias superclasses, é definida na subclasse como a conjunção dos esquemas que a definem em cada superclasse. Operações auxiliares, funções e tipos podem ser definidos nas cláusulas **state**, **initialstates** e **operations**.

A definição completa da linguagem, incluindo sua sintaxe formal e vários exemplos que ilustram sua semântica informal são encontrados em [14].

4.1.2 Processo de Desenvolvimento

Embora não exista um modelo do processo de desenvolvimento de software que seja universalmente aceito, um modelo simplificado mostrado em [21], pode ser visto como uma síntese. Segundo este modelo, a construção de qualquer software envolve duas atividades básicas: uma de *abstração* e outra de *satisfação*. O processo de desenvolvimento descrito nessa seção fornece um meio para efetivar estas duas atividades, sendo encarado como um detalhamento deste modelo simplificado.

Atividade de Abstração A atividade de abstração tem por objetivo definir claramente *o que* é o problema estudado, isto é, fornecer um modelo matemático que seja sua especificação. O interesse está no comportamento do sistema que se deseja construir, sem preocupações em onde e como será implementado. É composta pelas etapas de análise de requisitos e especificação formal, descritas a seguir.

A análise de requisitos é a primeira tarefa a ser realizada e deve identificar as classes de objetos que formam o domínio da aplicação, bem como os serviços que cada uma deve prestar. Define um vocabulário comum que unifica as etapas do desenvolvimento e as pessoas envolvidas. Tem como resultado uma lista inicial das classes encontradas, seus atributos e uma descrição informal dos serviços que devem fornecer, servindo como ponto de partida para a etapa de especificação formal.

Durante a etapa de especificação formal as classes identificadas na etapa anterior são especificadas, procurando-se agrupá-las em hierarquias conceituais através de técnicas de classificação(generalização/especialização, agregação e aproximação). A especificação é a primeira referência precisa da aplicação e serve como exercício para o seu entendimento. O resultado mais importante desta fase deve ser um modelo suficientemente completo do sistema, a partir do qual se pode começar a projetar um programa.

Atividade de Satisfação A atividade de satisfação é responsável em *como* solucionar o problema estudado, isto é, consiste em criar um programa que seja um modelo da especificação. O interesse está em encontrar as estruturas computacionais de dados e os algoritmos em que o sistema previamente especificado será implementado. Esta atividade está dividida nas etapas de projeto e de implementação, descritas a seguir.

No projeto, a idéia central é descrever as classes concretas que o programa usará para representar as classes de mais alto nível de abstração da especificação formal. Este processo, denominado *refinamento de classes*, está baseado no refinamento de dados apresentado em [19]. Refinamento de classes é o processo de mostrar o conjunto das operações definidas numa classe implementado numa subclasse por outro conjunto de operações num espaço de estado diferente. Permite que as classes abstratas que representam tipos de dados matemáticos sejam trocadas por subclasses mais concretas, orientadas a linguagem de programação usada para implementação. Por este processo, partindo da especificação formal dos requisitos, cada passo do refinamento resulta numa especificação menos abstrata que representa a anterior, só que com mais detalhes de implementação. Finalmente, a partir da especificação mais concreta, um programa é implementado. O resultado é uma seqüência de documentos de projeto, cada um descrevendo uma pequena coleção de decisões de projeto. O processo de refinamento é dito *direto* quando é possível ir da especificação formal mais abstrata para o programa final em apenas um passo. Fica evidente que é difícil demonstrar que o programa implementado é consistente com os requisitos do sistema. No entanto, isto pode ser feito pela verificação do projeto em cada passo. Quando esta verificação é formalmente provada, o desenvolvimento é *formal*, caso contrário, o desenvolvimento é *rigoroso*. Neste trabalho, o sistema médico proposto foi desenvolvido de maneira rigorosa e com refinamento direto.

A etapa de implementação é a responsável pela codificação do programa a partir da especificação com mais baixo nível de abstração fornecida pela fase de projeto. Deve levar em consideração aspectos como eficiência e o ambiente operacional.

A metodologia usada não empregou o refinamento de operação ou algorítmico, que consiste em mostrar que uma operação é implementada por outra no mesmo estado de espaço. Na sua forma geral, permite que construtores de uma linguagem de programação sejam gradativamente introduzidos no projeto.

4.2 Análise de Requisitos e Especificação

A especificação foi construída a partir da definição das classes mais básicas até as mais elaboradas, caracterizando o desenvolvimento "bottom-up" (composicional) da orientação a objetos.

PDF compression, OCR

Signal AnalogSignal Clock SampledSignal ECG CardiacCycle
DigitalFilter
EventDetector QRSDetector FiducialMarkDetector
Conversor StressTestConversor
Channel PhysicalChannel VirtualChannel VisualChannel ECGChannel
Exam StressExam
Measures
Person Patient Doctor
Protocol
MedicalSystem StressTestSystem

Tabela 1: Hierarquia das principais classes que formam a especificação

A tabela 1 mostra a hierarquia das principais classes da especificação do sistema. Durante a construção da especificação, buscou-se ter não só o sistema definido, mas também um arcabouço de especificações de classes que possam ser re-usadas e estendidas para novos desenvolvimentos neste domínio de aplicação e para aperfeiçoamento e melhor entendimento de outros sistemas anteriormente construídos por nós [4, 5] e para os quais não foi usada uma metodologia de desenvolvimento com base formal.

Neste trabalho, vamos mostrar como exemplo parte da especificação, projeto e implementação da classe *SampledSignal*, fundamental para todo o sistema. O objetivo é dar uma idéia do processo de desenvolvimento usado para construir o programa que implementa o SMBC, que atualmente tem cerca de 20.000 linhas de código e 93 classes, resultante de 90 páginas de especificação formal em MooZ que conta com 32 classes. Para o restante das classes da tabela 1, será dada uma definição informal para se ter uma visão global do sistema.

A classe *Signal* é um ancestral comum dos diversos tipos de sinais do domínio da aplicação, sendo modelado por uma classe que define uma função parcial do tempo para voltagem.

Class *Signal*

superclasses $PartialFunction(X \setminus TIME, Y \setminus VOLTS)[rel \setminus fun]$

Subclasse da classe genérica *PartialFunction*[1], instancia o tipo indefinido X (abscissas) para o tipo $TIME$ e o tipo indefinido Y (ordenadas) para $VOLTS$. O componente do estado da superclasse, *rel*, que modela uma função como um conjunto de pares $(TIME, VOLTS)$, é renomeado para *fun*, por ser um nome mais adequado para designar uma função. O domínio do tempo é representado pelos números reais não negativos e a voltagem é representada pelos reais. São definidos também os tipos para freqüência e período, usados em definições nas subclasses de *Signal*. Esta classe se propõe a ser apenas uma classe mais geral, que abstrai os mais variados tipos de sinais existentes e que são efetivamente úteis.

$$TIME == \mathbb{R}_0^+ \qquad VOLTS == \mathbb{R}$$
$$FREQUENCY == \mathbb{R}^+ \qquad PERIOD == \mathbb{R}^+$$

[1]Esta classe faz parte da biblioteca de classes primitivas de MooZ, que define uma valiosa variedade de construtores de tipo de definições matemáticas.

EndClass *Signal*.

AnalogSignal modela um sinal analógico e é uma subclasse de *Signal*. Especifica que todos os elementos do seu domínio devem ser definidos, correspondendo a uma função total do tempo para voltagem. A classe *Clock* especifica o relógio interno usado pelo conversor A/D, que é representado na especificação pela classe *Conversor*.

A classe *SampledSignal* é uma das mais importantes do sistema e especifica os sinais amostrados pelo conversor A/D. Um sinal amostrado, além de ser uma função discreta do tempo para voltagem, isto é, subclasse de *Signal*, também é subclasse de *FinitePartialFunction*, caracterizando que um sinal amostrado é finito. Possui operações que calculam sua duração, valores máximos, mínimos, transformada de Fourier, espectro de potência, testes de limiar, etc.

Class *SampledSignal*

superclasses $Signal, FinitePartialFunction(X \setminus TIME, Y \setminus VOLTS)[rel \setminus fun]$

Os componentes f_s e p_s são introduzidos no estado da classe e representam a freqüência e o período de amostragem do sinal, respectivamente.

state

$$
\begin{array}{|l}
\hline
f_s : FREQUENCY \\
p_s : PERIOD \\
\hline
f_s = 1/p_s \\
\forall t, t' : dom(fun) \bullet \\
\quad \exists n : \mathbb{N} \bullet |t - t'| = n * p_s \\
\#dom(fun) * p_s = duration(fun, p_s) \\
\hline
\end{array}
$$

O invariante primeiro estabelece que a freqüência e o período de amostragem são inversamente proporcionais. Depois afirma que todos os valores de tempo em que a função está definida são múltiplos do período de amostragem. Finalmente, determina que o sinal amostrado é completo, isto é, que está definido para todos os valores de tempo múltiplos de p_s e intermediários entre o tempo inicial e final da amostragem. Por exemplo, suponha que a amostragem de um sinal começou com tempo inicial $t_i = 12s$ e terminou no tempo $t_f = 32s$ com um período de amostragem de 5 segundos. O invariante garante que os tempos intermediários 17, 22 e 27 existem e que o domínio da função é $\{12, 17, 22, 27, 32\}$.

initialstates

A operação de inicialização da classe, *Init*, determina a freqüência de amostragem do sinal. Como esta classe é, em último caso, uma subclasse de *Relation*[2], há uma conjunção dos esquemas *Init* das duas classes, segundo a semântica de MooZ. Na conjunção de dois ou mais esquemas, as suas declarações são combinadas e ocorre a conjunção dos seus predicados. Deste modo, também está especificado que um sinal amostrado é vazio no seu estado inicial. A pré-condição da operação é que a freqüência de amostragem passada como parâmetro seja maior que zero.

Init

$$
\begin{array}{|l}
\hline
\Delta(f_s, p_s) \\
f_s? : FREQUENCY \\
\hline
f_s? > 0 \\
f_s' = f_s? \\
\hline
\end{array}
$$

operations

A operação *duration* retorna a duração de tempo do sinal amostrado.

[2]Uma função é subclasse de uma relação.

$$
\begin{array}{|l}
duration: \quad fun : \mathbb{P}(TIME \times VOLTS) \times p_s : PERIOD \rightarrow TIME \\
\hline
fun = \varnothing \Rightarrow duration(fun, p_s) = 0 \\
fun \neq \varnothing \Rightarrow duration(fun, p_s) = max(dom(fun)) - min(dom(fun)) + p_s
\end{array}
$$

A operação *timeAveraging* retorna um sinal em que cada valor corresponde à média dos valores existentes posteriormente numa janela de tempo. A pré-condição da operação é que esta janela de tempo seja múltipla do período de amostragem. Esta operação presta serviço à técnica de detecção de QRS por nós especificada e que está definida informalmente em [8].

$$
\begin{array}{|l}
timeAveraging: \quad fun : \mathbb{P}(TIME \times VOLTS) \times p_s : PERIOD \times \\
\qquad\qquad TIME \nrightarrow fun : \mathbb{P}(TIME \times VOLTS) \\
\hline
\forall t_w : TIME \mid \exists n : \mathbb{N}_1 \bullet n * p_s = t_w \bullet \\
\quad timeAveraging(fun, p_s, t_w) = \lambda\, t : dom(fun) \mid t \leq max(dom(fun)) - t_w + p_s \bullet \\
\qquad\qquad \frac{p_s}{t_w} * \sum i : t..t + t_w - p_s \bullet fun(i)
\end{array}
$$

EndClass *SampledSignal*.

A classe *ECG*, subclasse de *SampledSignal*, armazena a seqüência dos tempos em que ocorrem os complexos QRS e operações para cálculo da freqüência cardíaca, do intervalo RR mediano, do ciclo cardíaco mediano, etc. A classe *CardiacCycle* armazena uma tabela dos tempos em que ocorrem as ondas P,Q,R,S,T, início e fim do QRS e da onda T. Diversas operações estão definidas, como a que corrige o desvio da linha de base e as que calculam a duração do complexo QRS; integral do segmento ST; amplitudes de Q, R, S e T; intervalo QT; ST-60 e 80; integral do segmanto ST; índice ST; etc.

A classe *EventDetector* é o ancestral comum dos diversos tipos de detetores de eventos, possuindo uma operação *Detect* que recebe um sinal amostrado como parâmetro e o devolve com os eventos localizados. *QRSDetector* e *FiducialMarkDetector* são algumas das suas subclasses e são ancestrais de diversos detetores de QRS e de marcas fiduciais (localizações das ondas P, Q, R, S, T, etc), respectivamente. O primeiro interage com objetos da classe *ECG* e o segundo com objetos da classe *CardiacCycle*.

A classe *Conversor* é um modelo matemático abstrato de um conversor analógico-digital (A/D) de n canais. Possui um clock que define a freqüência de amostragem e um conjunto de canais da classe *Channel* ou de suas subclasses. Pode trabalhar no modo de monitorização, utilizando-se canais visuais que mostram continuamente os sinais convertidos. Os canais são totalmente independentes, podendo trabalhar com escalas de tempo e amplitude diferentes, que podem ser modificadas durante a monitorização. A classe *StreesTestConversor* define o conversor utilizado para monitorização e aquisição dos sinais de ECG durante o teste de esforço. Possui três canais do tipo *ECGChannel* que monitoram continuamente três derivações de ECG do paciente, detectando o complexo QRS para cálculo on-line da FC. Esta classe está definida para trabalhar de acordo com um protocolo da classe *Protocol*, onde estão definidas as diversas fases do exame. Ao final de cada fase do protocolo, são armazenados n segundos de sinal e a pressão arterial do paciente, que serão processados posteriormente pelo detetor de QRS para obtenção dos ciclos cardíacos medianos de cada derivação.

A classe *MedicalSystem* é o ancestral comum dos diversos tipos de sistemas médicos. Possui três tabelas contendo objetos das classes *Patient*, *Doctor* e *Exam* e operações para o seu gerenciamento (inclusão, remoção, atualização, pesquisa, etc). O sistema de teste de esforço, *StressTestSystem*, é uma subclasse de *MedicalSystem*. Adicionalmente ao comportamento e estrutura de sua superclasse, possui uma tabela de protocolos que pode ser configurada pelo usuário e um objeto da classe *StressTestConversor*. Especifica que a sua tabela de exames só deve ter objetos da classe *StressTestExam*, onde os sinais convertidos para cada estágio do protocolo usado são armazenados para processamento off-line posterior. O processamento do sinal envolve a deteção do QRS pelo detetor de QRS, cálculo dos ciclos cardíacos medianos, determinação das marcas fiduciais pelo detetor de marcas fiduciais, correção da linha de base e cálculo dos parâmetros de interesse do ECG de esforço. Um relatório final contendo as medidas

PDF compression, OCR, web optimization using a watermarked evaluation copy

calculadas automaticamente e o diagnóstico digitado pelo médico é emitido pelo sistema. O relatório inclui as medianas de todas as derivações de cada estágio.

4.3 Projeto e Implementação

A especificação formal do SMBC apresentou uma descrição das classes que formam o sistema em termos de estruturas matemáticas de dados, altamente abstratas, como na classe *SampledSignal*, definida por uma função parcial finita. O interesse estava em definir de forma precisa e clara as suas principais propriedades, sem preocupação de como ou onde seriam implementadas. No projeto, em contraposição, procurou-se mostrar de que forma estas classes da especificação abstrata foram implementadas usando estruturas computacionais de dados. A preocupação maior estava em criar os programas que satisfizessem as propriedades especificadas anteriormente.

O sistema foi implementado em TURBO PASCAL[2], versão 6.0, extensão modular e orientada a objetos da linguagem de programação Pascal. As estruturas de dados da linguagem de programação foram modeladas em MooZ por tipos de dados matemáticos com o objetivo de possibilitar o estabelecimento de uma relação de abstração entre a especificação formal de mais alto nível e o programa que a implementa. Esta relação, no caso do refinamento direto usado neste desenvolvimento, foi estabelecida pela especificação concreta, que pode ser vista como um modelo tanto do programa, quanto da especificação abstrata. A representação em MooZ das estruturas de dados da linguagem de progamação e os motivos que justificaram sua escolha estão presentes em [17]. A seguir vamos descrever parte do projeto e da implementação da classe *SampledSignal*, especificada anteriormente.

A classe *CSampledSignal* descreve a representação concreta usada para a classe *SampledSignal*. Um sinal amostrado, anteriormente modelado como uma subclasse de uma função parcial finita, foi representado na linguagem de programação como uma subclesse de uma seqüência de inteiros. Esta seqüência é definida pela classe genérica *Collection*, que especifica em MooZ a classe `TCollection` da biblioteca de objetos de TURBO PASCAL 6.0. `TCollection` implementa uma seqüência de itens que pode crescer dinamicamente. O mecanismo de herança foi usado para permitir o relacionamento entre o modelo abstrato e o concreto da classe.

Class *CSampledSignal*

superclasses $SampledSignal, Collection(ITEM \setminus \mathbb{Z})[Init \setminus CInit]$

A instanciação do item na coleção para o tipo $\mathbb{Z}$ se deve ao fato que o conversor A/D físico quantifica para inteiros os valores de voltagem convertidos, de acordo com sua resolução.

state

O estado da classe introduz as representações concretas para a freqüência e o período de amostragem. O invariante estabelece que o período é o inverso da freqüência de amostragem.

$$\begin{array}{|l} cf_s : FREQUENCY \\ cp_s : PERIOD \\ \hline cf_s = 1/cp_s \end{array}$$

A idéia desta representação é que, para cada tempo em que foi amostrado um valor de voltagem, exista uma posição na coleção que armazene um inteiro correspondente a esta voltagem e vice-versa. Isto está documentado pelo esquema *Abs*, que define a relação de abstração entre o estado da classe abstrata *SampledSignal* e o estado da classe concreta *CSampledSignal*.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

Abs

$\Xi(fun, f_s, p_s, items, count, cp_s, cf_s)$

$cf_s = f_s$

$cp_s = p_s$

$\#dom(fun) = count$

$fun = \{t : TIME;\ v : VOLTS;\ n : loLimit..loLimit + count - 1 \mid$

$voltsToInt(v) = items(n) \land t = (n - loLimit) * p_s + min(dom(fun)) \bullet t \mapsto v\}$

As duas primeiras linhas afirmam que a freqüência e o período de amostragem concretos são iguais a freqüência e período de amostragem abstratos, respectivamente. Depois, afirma que o número de pontos amostrados na classe abstrata é igual ao número de elementos, *count*, da coleção. Por fim, estabelece uma relação entre uma posição e valor na coleção a um tempo e voltagem na função que representa um sinal amostrado na classe abstrata. A função *voltsToInt* realiza a quantificação de um valor de voltagem para um inteiro. *items* é a seqüência de inteiros da coleção e *loLimit* é uma constante que define o índice do primeiro elemento da coleção.

A partir deste esquema de abstração, pode-se mostrar alguns fatos interessantes. Por exemplo, a determinação do índice n na coleção que corresponde a um tempo t de amostragem no sinal é mostrado a seguir:

$$
\begin{array}{lll}
t & = (n - loLimit) * p_s + min(dom(fun)) & [Abs] \\
min(dom(fun)) + m * p_s & = (n - loLimit) * p_s + min(dom(fun)) & [inv.\ de\ SampledSignal] \\
m * p_s & = (n - loLimit) * p_s & [min(dom(fun)) - min(dom(fun)) = 0] \\
m & = (n - loLimit) & [p_s / p_s = 1] \\
n & = loLimit + m & [reorganizando\ os\ termos]
\end{array}
$$

Quando t é igual ao tempo inicial de amostragem, tem-se que $t = min(dom(fun)) + m * p_s$, onde $m = 0$. Logo, a posição na coleção que corresponde ao tempo inicial de amostragem é *loLimit*. Já a posição n na coleção para o tempo final de amostragem, quando o sinal não é vazio, corresponde à posição $loLimit + count - 1$. Isto é, a posição que corresponde ao tempo $max(dom(fun))$ no sinal é indexada por $loLimit + count - 1$, que corresponde ao último elemento da coleção.

Essa classe concreta em MooZ, além de ser um modelo para a classe abstrata *SampledSignal*, também é um modelo para a classe `SampledSignal` em TURBO PASCAL 6.0, que implementa um sinal amostrado. A definição parcial dessa classe na linguagem programação e dos tipos usados é mostrada a seguir:

```
Type
    Time      = Real;          Volts  = Real;
    Frequency = Real;          Period = Real;

SampledSignal = object(TCollection(Integer))
                  fs: Frequency;
                  ps: Period;
  Constructor Init(limitI : LimitBoundaries; deltaI: SizeBoundaries; fsI : Frequency);
  Function    Duration: Time;
  Function    TimeAveraging(pw: LimitBoundaries): SampledSignal;
end;
```

initialstates

A operação *CInit* define a inicialização da classe concreta. Nesta operação, está implícita a conjunção com a operação *CInit* da superclasse *TCollection*. A pré-condição é que a freqüência de amostragem passada como parâmetro seja maior do que zero.

CInit

$\Delta(cf_s, cp_s)$

$f_s? : FREQUENCY$

$f_s? > 0$

$cf_s' = f_s?$

Para um desenvolvimento formal, deveria ser provado que cada possível estado inicial definido pela classe concreta representa um possível estado inicial definido pela classe abstrata, isto é:

$$\forall\, CState' \bullet CInit \Rightarrow (\exists\, AState' \bullet Init \wedge Abs')$$

onde *CState'* e *AState'* correspondem aos estados das classes concreta e abstrata após a realização das operações *Cinit* e *Init*, respectivamente. *Abs'* corresponde a relação de abstração entre as classes concreta e abstrata após as sua respectivas operações de inicialização. Entretanto, como este desenvolvimento foi rigoroso, esta e outras provas não foram feitas, apesar de sua realização não impor maiores problemas conceituais no desenvolvimento.

Esta especificação do estado inicial concreto foi usada para escrever a operação de inicialização na linguagem de programação. Observa-se que a disciplina do uso de pré-condições na definição da operação permite um controle sobre as suas condições de sucesso ou fracasso.

```
Constructor SampledSignal.Init;
begin
  TCollection.Init(limitI,deltaI);
  if fsI > 0 then begin
    fs := fsI;
    ps := 1/fsI;
  end
  else
    Error(1);
end;
```

operations

A operação concreta *cDuration* redefine a operação *Duration* da classe abstrata *SampledSignal*.

$$\begin{array}{|l} cDuration: \quad count : 0..maxCollectionSize \times cp_s : PERIOD \rightarrow TIME \\ \hline duration(count, cp_s) = count * cp_s \end{array}$$

Pode ser facilmente mostrado usando o invariante de *SampledSignal*, a definição de *duration* e o esquema de abstração *Abs* que a tempo de duração do sinal em segundos é igual a $count * cp_s$. Esta operação é implementada em TURBO PASCAL 6.0 por uma função do seguinte modo:

```
Function SampledSignal.Duration;
begin
  Duration := count * ps;
end;
```

A operação *cTimeAveraging* redefine a operação *timeAveraging* da classe abstrata, onde a janela de tempo t_w corresponde nesta operação a p_w, um número de posições. Esta operação retorna uma nova coleção, com tamanho $count - p_w + 1$, que corresponde à media temporal do sinal receptor da mensagem.

$$\begin{array}{|l} cTimeAveraging: \quad items : Map(loLimit..hiLimit, \mathbb{Z}) \times count : 0..maxCollectionSize \times \\ \qquad\qquad 1..maxCollectionSize \nrightarrow \\ \qquad\qquad (items : Map(loLimit..hiLimit, Code) \times count : 0..maxCollectionSize) \\ \hline \forall\, p_w : 1..maxCollectionSize \bullet \\ \quad \exists\, items' : Map(loLimit..hiLimit, Code) \bullet \\ \quad\quad (\forall\, p : loLimit..loLimit + count - p_w \bullet \\ \quad\quad\quad items'(p) = \frac{1}{p_w} * \sum i : p..p + p_w - 1 \bullet items(i)) \wedge \\ \quad\quad cTimeAveraging(items, count, p_w) = (items', count - p_w + 1) \end{array}$$

Esta operação foi implementada pela função

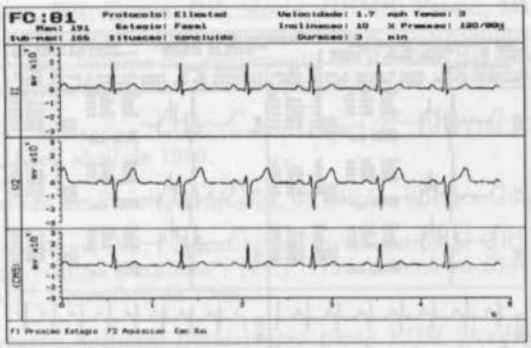

Figura 3: Tela de monitorização e aquisição com cálculo em tempo real da freqüência cardíaca

```
Function SampledSignal.TimeAveranging;
var
   p,i    : Integer;
   somat  : LongInt;
   sig    : SampledSignal;
begin
  sig.Init(limit,delta,fs);
  for p := LoLimit to LoLimit + count - pw do begin
    somat := 0;
    for i := p to p + pw - 1 do
      somat := somat + items^[i];
    sig.Insert(somat div pw);
  end;
  TimeAveraging := sig;
end;
```

EndClass *CSampledSignal*.

5 Resultados

O protótipo do sistema desenvolvido é de boa qualidade e foi construído num tempo excepcionalmente curto (6 meses) para sua complexidade. Demonstrou excelente desempenho, mesmo funcionando num microcomputador PC-XT, 12MHz. Com o objetivo de avaliar o funcionamento do sistema implementado, foi realizado um conjunto preliminar de vinte exames em indivíduos normais no Serviço de Ergometria do Setor de Cardiologia do Hospital Universitário da UFPB. Os parâmetros de interesse medidos pelo sistema refletiram que os indivíduos que se submeteram aos exames eram normais. A figura 3 exemplifica um objeto da classe *StressTestConversor*, na fase de monitorização, durante um teste de esforço, com o cálculo em tempo real da FC. A figura 4 mostra a tela gráfica de parte do relatório relativo aos dois primeiros estágios de um exame, com os ciclos cardíacos medianos e parâmetros de interesse calculados. Atualmente o sistema está em processo de validação clínica, com exames realizados em pacientes, visando certificá-lo para permitir a transferência desta tecnologia para uma empresa brasileira que demonstrou interesse em adquiri-la.

6 Conclusões

A aplicação de especificações formais orientadas a objetos possibilitou uma boa concepção e projeto do sistema e possivelmente evitou erros durante a sua implementação. Isto porque o

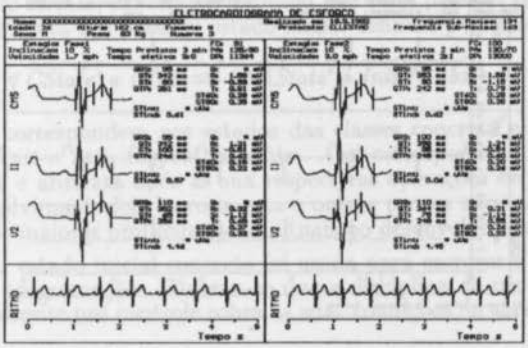

Figura 4: Relatório gráfico do exame com as medianas e parâmetros de interesse calculados

uso de uma notação matemática abstrata, estruturada e modular deu o suporte necessário para um bom entendimento do problema estudado, mesmo quando complexo, nos estágios iniciais do desenvolvimento. Propiciou, ainda, uma documentação precisa, não ambígua e independente de uma implementação ou de um ambiente operacional específicos, permitindo deste modo uma maior facilidade de intercâmbio do conhecimento neste domínio de aplicação.

Isto está evidenciado pela especificação formal do sistema, onde estão descritas de forma precisa suas características. Durante a construção desta especificação, propriedades do sistema puderam ser entendidas antes que fossem implementadas. Algumas delas foram modificadas apenas com base na especificação, antes mesmo que se tivesse uma implementação, onde normalmente os custos para alterações são mais altos. Por outro lado, alguns aspectos do sistema só foram mudados após a sua implementação, mas mesmo assim, o uso de uma notação formal mostrou-se benéfico porque tais mudanças eram sempre feitas baseadas na especificação formal, onde o nível de abstração é mais alto. Como conseqüência, permitiu o entendimento mais fácil do que ocorreu no restante do sistema em função destas mudanças, possibilitando uma implementação mais segura do software.

Podemos concluir, depois da realização deste trabalho, que a aplicação rigorosa de especificações formais orientadas a objetos é, nos dias atuais, tecnicamente viável, podendo ser utilizada em desenvolvimentos reais de software. Entretanto, para que seu uso se torne efetivo, é preciso ter profissionais capacitados e ferramentas que auxiliem na escrita e gerenciamento de grandes especificações.

Referências

[1] Alencar, AJ; Goguen, JA. "OOZE: An Object Oriented Z Environment". *In Proceedings of ECOOP'91 - V European Conference on Object Oriented Programing*, Springer-Verlag, Genebra - Suíça, 1991.

[2] Borland International. *Turbo Pascal Version 6.0 — Programmer's Guide*, 1990.

[4] Carvalho, LC; Barros, RV; Lima, AP; Motta, GHMB; Machado, PDL. "A Versatile Catheterization Laboratory Data Management System Based on PC-XT Microcomputers". *Medical and Biological Engineering and Computing*, volume 29, supplement part 2. Digest of the World Congress on Medical Physics and Biomedical Engineering, Kyoto, Japão, 1991.

[5] Carvalho, LC; Maia, RR; Torres, RH; Motta, GHMB; Varani, ML. "Processamento Automático de Sinais Eletrocardiográficos com o uso de Microcomputadores". *Arquivos Brasileiros de Cardiologia*, volume 59, suplemento II. Resumo das Comunicações do XLVIII Congresso da Sociedade Brasileira de Cardiologia, Recife - PE, setembro de 1992.

[6] Duke, D; Duke, P. "Towards a Semantics for Object-Z". *In VDM'90: VDM and Z!, LNCS*, Springer-Verlarg, Kiel, abril de 1990.

[7] Guyton, AC. *Tratado de Fisiologia Médica*, 6ª edição, Editora Guanabara, 1986.

[8] Hamilton, PS; Tompkins, WJ. "Quantitative Investigation of QRS Detection Rules Using the MIT/BIH Arrhythmia Database". *IEEE Transactions on Biomedical Engineering*, volume 33, número 12, dezembro de 1986.

[9] Jones, CB. *Systematic Software Development Using VDM*, 2ª edição, Prentice Hall International Series, Prentice Hall, 1990.

[10] Kriewall, TJ; Long, JM. "Computer-Based Medical Systems". *IEEE Computer*, volume 24, número 3, março de 1991.

[11] Lano, K. "Z++: An Obejct Oriented Extension to Z". *Z Users Meeting*. Workshop on Computing Science, Springer-Verlag, Oxford UK, dezembro de 1990.

[12] Marcondes, GD. *Ergometria/Bases da Reabilitação Cardiovascular*, Cultura Médica, 1986.

[13] Meira, SL; Cavalcanti, ALC. "Modular Object-Oriented Z Specifications". *Z Users Meeting*. Workshop on Computing Science, Springer-Verlag, Oxford-UK, dezembro de 1990.

[14] Meira, SL; Cavalcanti, ALC. "The MooZ Specification Language". *ProTeM-CC-NE, Relatório Técnico ES/1.92*, Universidade Federal de Pernambuco, janeiro de 1992.

[15] Meira, SL; Sampaio, AC. "Modular Extensions to Z". *In VDM'90: VDM and Z!, LNCS*, Springer-Verlarg, Kiel, abril de 1990.

[16] Milner, R. "Is Computing an Experimental Science?". *Laboratory for Foundations of Computer Science Report Series, ECS-LFCS-86-1*, Department of Computer Science, University of Edinburgh, agosto de 1986.

[17] Motta, GHMB. *Especificações Formais Orientadas a Objetos: Aplicação no Desenvolvimento de um Sistema para Processamento do Eletrocardiograma de Esforço*, Dissertação de Mestrado, Universidade Federal de Pernambuco, Departamento de Informática, Recife - PE, 1992.

[18] Motta, GHMB; Carvalho, LC; Meira, SL. "Especificações Formais Orientadas a Objetos: Aplicação no Processamento do Eletrocardiograma de Esforço". *Anais do I Forum Nacional de Ciência e Tecnologia em Saúde: XIII Congresso Brasileiro de Engenharia Biomédica*, Hotel Glória, Caxambu-MG, 20 a 24 de novembro de 1992.

[19] Spivey, JM. *The Z Notation: A Reference Manual*, Prentice Hall, 1989.

[20] Stepney, S; Barden, R; Cooper, D. "Comparative Study of Object Orientation in Z". *Technical Report 1*, Logica Cambridge Limited, Advanced Software Engineering Division, fevereiro de 1991.

[21] Turski, WM; Maibaum, TSE. *The Specifications of Computer Systems*. International Computer Science Series, Adison-Wesley Publishing Company, 1987.

[22] Wegner, P. "Concepts and Paradigms of Object-Oriented Programing. Expansion of Oct 4 OOPSLA-89 Keynote Talk". *ACM Object-Oriented Messenger*, volume 1, número 1, 1990.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor