Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática

Qualidade e Confiabilidade de Software

Projeto #1

QCS 2014/2015

Web services e programação N-Versões

André P. Gonçalves, Jóni Oliveira e José M. Alves
{apcosta, joniro, jmcalves}@student.dei.uc.pt

17 de Maio de 2015

Índice

1. **Introdução**

A programação N-Versões consiste em desenvolver N *softwares* independentes a partir da mesma especificação. Os *softwares*, apesar de desenvolvidos separadamente, e por norma equipas diferentes, é semelhante na sua funcionalidade, apresentando o mesmo *output* quando os *inputs* são iguais nas N versões. O objetivo é comparar o resultado obtido nas diferentes versões e estabelecer um *output* final sendo o comum entre as diferentes versões, de forma a minimizar situações de erro no *software* e assegurar um *output* correto. Usualmente este tipo de programação é usada em sistemas críticos em que um resultado errado poderá resultar em inúmeros danos (ex: aviação).

O objetivo deste trabalho é desenvolver um *webservice* baseado em programação N-versões para calcular a dose de insulina em pacientes diagnosticados com Diabetes Tipo-II. Para o realizar, cada grupo irá criar a sua versão do *webservice* com base na mesma especificação e utilizará os *webservices* desenvolvidos pelos restantes grupos para fornecer os vários *inputs* ao votador. Para interagir com o sistema irá ser desenvolvido um pequeno *website* que interage com este votador e disponibiliza o resultado na página.

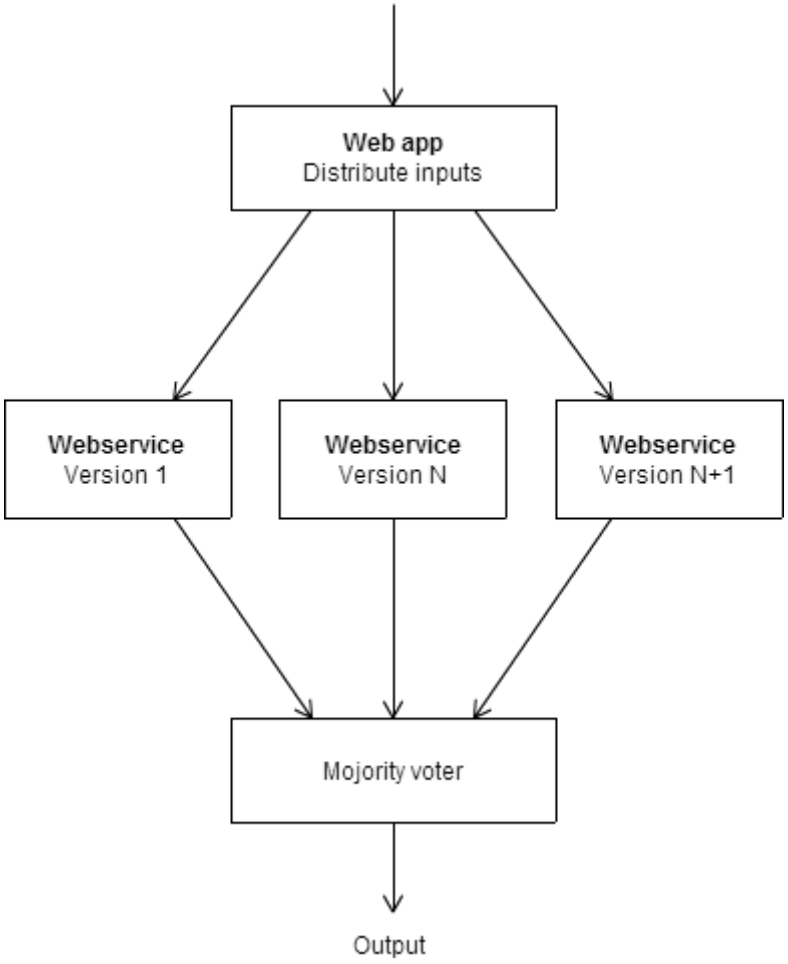

Figura 1. Esquema de funcionamento de programação N-versões

2. Especificação

2.1. Webservice para o cálculo da dose de insulina

O *webservice* implementado possui uma classe (InsulinDoseCalculator) e o seu funcionamento é descrito de seguida.

- **Método: *mealtime_insulin_dose***

  Função: calcula o número de unidades de insulina necessárias após uma refeição.

  Retorna: número de unidades de insulina de acção rápida necessárias após a refeição (i.e., bolus insulin replacement dose).

  Inputs:

1. total de gramas de carbohidratos na refeição (tipicamente entre 60g e 120g);
2. total de gramas de carbohidratos processados por 1 unidade de insulina de ação rápida (tipicamente 12g);
3. atuais níveis de açúcar no sangue medidos antes da refeição (tipicamente acima de 100 mg/dl);
4. níveis de açúcar no sangue desejados antes da refeição (tipicamente entre 80 e 120 mg/dl);
5. sensibilidade individual (1 unidade de insulina reduz os níveis de açúcar no sangue em 50 mg/dl).

- **Método: *background_insulin_dose***

  Função: calcula o número total de unidades de insulina necessárias entre refeições.

  Retorna: Dose de insulina basal.

  Input: Peso em quilogramas

- **Método: *personal_insulin_sensivity***

  Função: determina a sensibilidade de um indivíduo a uma dose de insulina.

  Retorna: a descida dos níveis de açúcar no sangue resultantes de uma unidade de insulina, em mg/dl.

  Inputs:

  1. nível de actividade física para actual dia (0-10);
  2. K amostras do níveis de actividade física em um dia (0-10);
  3. K amostras das descidas dos níveis de insulina a partir de 1 unidade de insulina nesse dia (25 a 100 mg/dl).

2.2. Requisitos funcionais

- **ID: FR1**

  Título: Seleccionar o tipo de dose de insulina a calcular e apresentar os resultados no ecrã a partir dos parâmetros de entrada.

  Descrição: O utilizador deve ser capaz de seleccionar o tipo de cálculo de dose de insulina, que deve ser escolhido a partir da lista seguinte:

  ➔ Mealtime insulin dose - standard insulin sensitivity
  ➔ Mealtime insulin dose - personal insulin sensitivity
  ➔ Background insulin dose

  Quando é feita uma selecção, os campos de entrada apropriados são apresentados. Adicionalmente, informação previamente inserida (se existir) deve ser eliminada.

  Rational: De forma ao utilizador poder seleccionar o tipo de dose de insulina a calcular.

  Dependências: Nenhuma

- **ID: FR2**

  Título: Cálculo da dose de insulina à refeição através da sensibilidade *standard* à insulina

  Descrição: O utilizador deve poder inserir os seguintes parâmetros nos campos apropriados:

  ➔ Total de gramas de carbohidratos na refeição; campo inicialmente vazio.
  ➔ Total de gramas de carbohidratos processados por 1 unidade de insulina de ação rápida; o campo contém por defeito o valor 12 mas pode ser modificado.
  ➔ Níveis de açúcar no sangue medidos antes da refeição; campo inicialmente vazio.
  ➔ Níveis de açúcar no sangue desejados antes da refeição; campo inicialmente vazio.

- ➔ Sensibilidade individual; o campo contém por defeito o valor 50 mas pode ser modificado.

Após inserir todos os parâmetros de entrada, o botão “Calcular dose de insulina” deve ficar ativo. Quando o utilizador pressiona o botão, o sistema deve calcular a dose de insulina à refeição para uma sensibilidade à insulina *standard*.

Rational: De forma a permitir ao utilizador inserir os parâmetros de entrada e obter a dose de insulina à refeição para uma sensibilidade à insulina *standard*.

Dependências: FR1

- **ID: FR3**

Título: Cálculo da dose de insulina à refeição através da sensibilidade pessoal à insulina

Descrição: O utilizador deve puder inserir os seguintes parâmetros nos campos apropriados:

- ➔ Total de gramas de carbohidratos na refeição; campo inicialmente vazio.
- ➔ Total de gramas de carbohidratos processados por 1 unidade de insulina de ação rápida; o campo contém por defeito o valor 12 mas pode ser modificado.
- ➔ Níveis de açúcar no sangue medidos antes da refeição; campo inicialmente vazio.
- ➔ Níveis de açúcar no sangue desejados antes da refeição; campo inicialmente vazio.
- ➔ Nível de atividade física para o presente dia; campo inicialmente vazio.
- ➔ Até 10 amostras dos níveis de atividade física num determinado dia.
- ➔ Até 10 amostras da redução dos níveis de açúcar no sangue a partir de uma unidade de insulina num determinado dia.

Após preencher todos os campos e o número de amostras é igual nos dois campos, o botão “Calcular dose de insulina” deve ficar activo. Quando o

utilizador pressiona o botão, o sistema deve calcular a dose de insulina à refeição para uma sensibilidade individual à insulina. A sensibilidade individual a uma unidade de insulina é calculada usando uma simples regressão linear a partir das amostras fornecidas.

Rational: De forma a permitir ao utilizador inserir os parâmetros de entrada e obter a dose de insulina à refeição para uma sensibilidade individual à insulina.

Dependências: FR1

- **ID: FR4**

  Título: Cálculo da dose de insulina basal

  Descrição: O utilizador deve poder inserir informação do campo apropriado:

  ➔ Peso em quilogramas; campo inicialmente vazio

  Após preencher todos os campos e o número de amostras é igual nos dois campos, o botão “Calcular dose de insulina” deve ficar ativo. Quando o utilizador pressiona o botão, o sistema deve calcular a dose de insulina basal.

  Rational: De forma a permitir ao utilizador inserir os parâmetros de entrada e obter a dose de insulina basal.

  Dependências: FR1

- **ID: FR5**

  Título: Mostrar detalhes técnicos do cálculo da dose de insulina

  Descrição: O utilizador deverá ser capaz de ver informações detalhadas sobre o cálculo da dose de insulina. Após o cálculo da dose de insulina (em todos os tipos de cálculo da dose de insulina definidos no FR1), o botão "Mostrar Informações Técnicas" deve ficar ativo. Quando o utilizador clicar no botão "Mostrar Informações Técnicas", o sistema deve mostrar as seguintes informações:

  ➔ Número de *webservices* (versões) usados para o cálculo.

- ➔ O resultado (dose de insulina final) calculada por cada versão de *webservice*.
- ➔ O resultado maioritário (resultado final produzido pelo votador) calculado a partir dos vários resultados individuais de cada *webservice*.

Se um webservice não produz um resultado válido (exemplo: o cálculo termina com um *timeout*), os detalhes técnicos disponíveis para essa versão devem mostrar "*Timeout*" em vez de um valor numérico (dose de insulina).

Rational: Este requisito funcional está relacionado com a restrição de design DR1 (descrita mais adiante) que indica que o sistema deve ser implementado usando programação N-versões. De forma ao utilizador poder consultar informação detalhada sobre o resultado produzido por cada versão do *webservice* e pelo votador.

Dependências: FR1, FR2, FR3, FR4

2.3. Requisitos não funcionais

A aplicação que irá ser usado pelo paciente para obter as doses de insulina deve preencher os seguintes requisitos não-funcionais:

- **ID: NFR1**

  Título: Cálculo da dose de insulina confiável

  Descrição: O resultado do cálculo dose de insulina apresentada ao utilizador deve ser confiável, para conseguir alcançar tal resultado o calculo deve ser feito por três ou mais entidades independentes e decidido por um votador por maioria, que tem de estar formalmente correto.

  Rational: Por forma ao utilizador receber um resultado confiável, se não for possível estabelecer uma maioria não deve ser mostrado nenhum resultado.

  Dependências: Nenhuma

- **ID: NFR2**

  Título: Performance do cálculo da dose de insulina

  Descrição: O sistema deve produzir o cálculo da dose de insulina em menos de 4 segundos, após o utilizador pressionar o botão “Calcular dose de insulina”.

  Rational: De forma a estabelecer um tempo máximo que o utilizador deve esperar pelo cálculo.

  Dependências: Nenhuma

- **ID: NFR3**

  Título: Tentar cálculos novamente

  Descrição: O sistema deve voltar a tentar o cálculo da dose de insulina, com os mesmos dados inseridos pelo utilizador, se não foi possível estabelecer uma maioria na votação. As próximas tentativas poderão usar diferentes *web services*, se disponíveis, e não deve demorar mais de 4 segundos.

  Rational: De forma a melhorar a disponibilidade do cálculo dentro do tempo limite estabelecido para o sistema fornecer resultados.

  Dependências: NFR2

2.4. Restrições no *design*

- **ID: DC1**

  Título: Programação N-Versões

  Descrição: O sistema deve usar Programação N-Versões

  Dependências: Nenhuma

- **ID: DC2**

  Título: Tecnologia *web services*

Descrição: O sistema deve usar *web services* para calcular a dose de insulina e cada versão dos *web services* deve executar em máquinas diferentes.

Dependências: DC1

- **ID: DC3**

  Título: Validação formal do votador usando *model checking*

  Descrição: O votador especificado e desenvolvido em programação N-Versões deve ser exaustivamente verificado através de *model checking*.

  Dependências: DC1

- **ID: DC4**

  Título: Mecanismo de votação

  Descrição: O votador deve usar um algoritmo simples que determine a maioria de valores inteiros relativamente à dose de insulina devolvida pelos vários *webservices*. Dois resultados por duas versões devem ser considerados equivalentes se a sua diferença é menor ou igual a 1 (dose de insulina). O resultado da votação deverá ser o resultado maioritário ou um código de erro se não existir maioria.

  Dependências: DC1, DC2

- **ID: DC5**

  Título: Precisão dos cálculos e arredondamento

  Descrição: Os métodos devem representar os valores numéricos usando o tipo de dados em Java *double* e o arredondamento para converter o resultado para inteiro deve ser realizado após todas os cálculos intermédios e usando as regras de arredondamento *standard.*

  Dependências: FR2, FR3 and FR4

3. Arquitetura do *software*

3.1. Geral

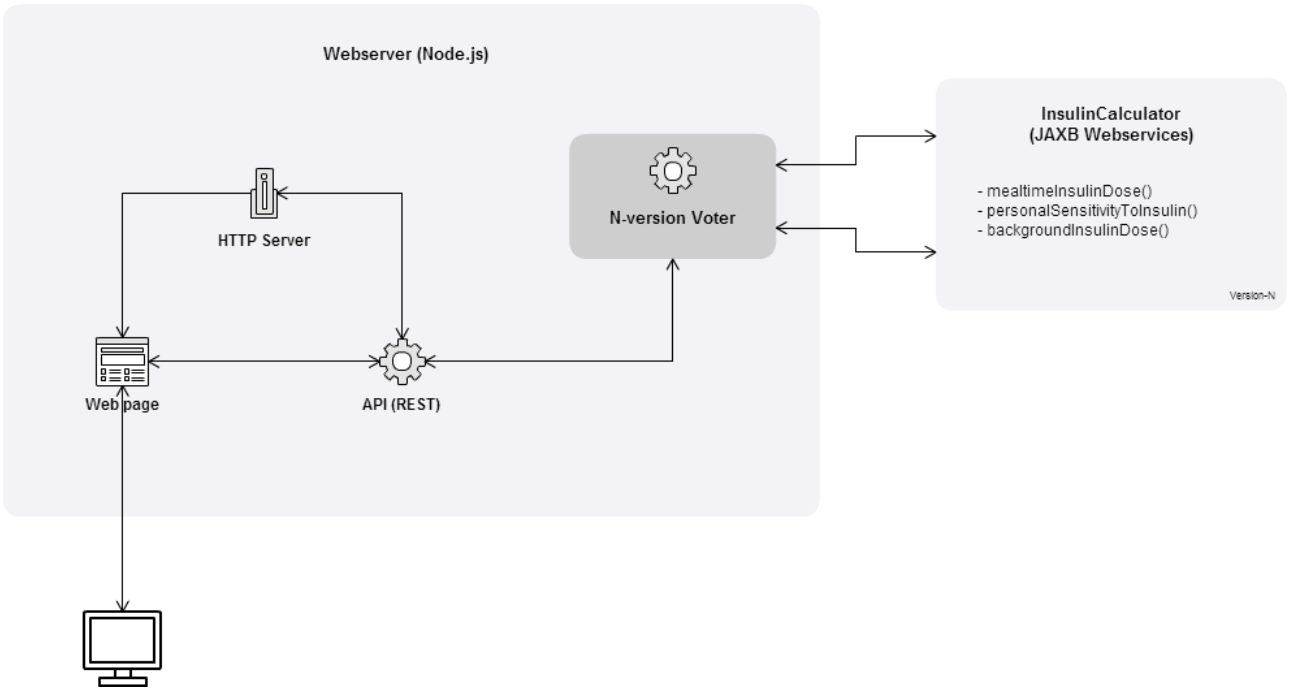

Figura 2. Diagrama de Componentes

Existem 2 componentes principais na arquitetura:

- Calculadora de insulina (*Webservices JAXB*)
- *Web server* (Votador + interface *web*)

O projeto contêm dois servidores, *QCS Server* e *Insulin App Server*, o primeiro contêm os *webservices*, no ponto 3.2 poderá verificar as suas especificações e componentes, o segundo contêm a aplicação *web* e o votador, no ponto 3.2 poderá verificar as suas especificações.

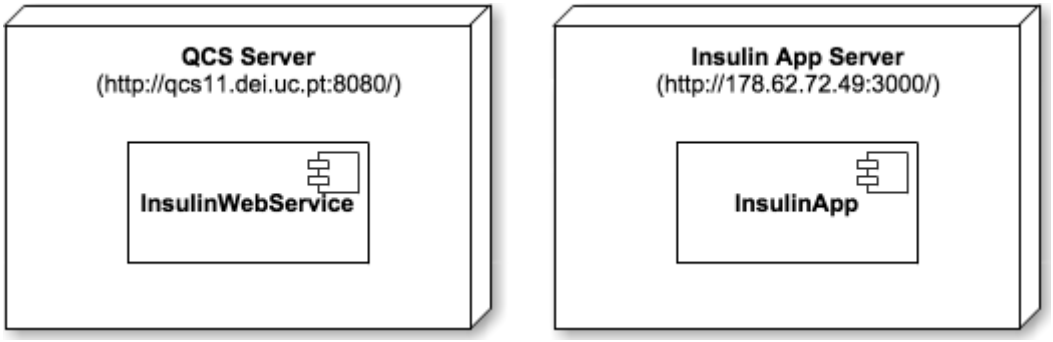

Figura 3. Diagrama de Componentes

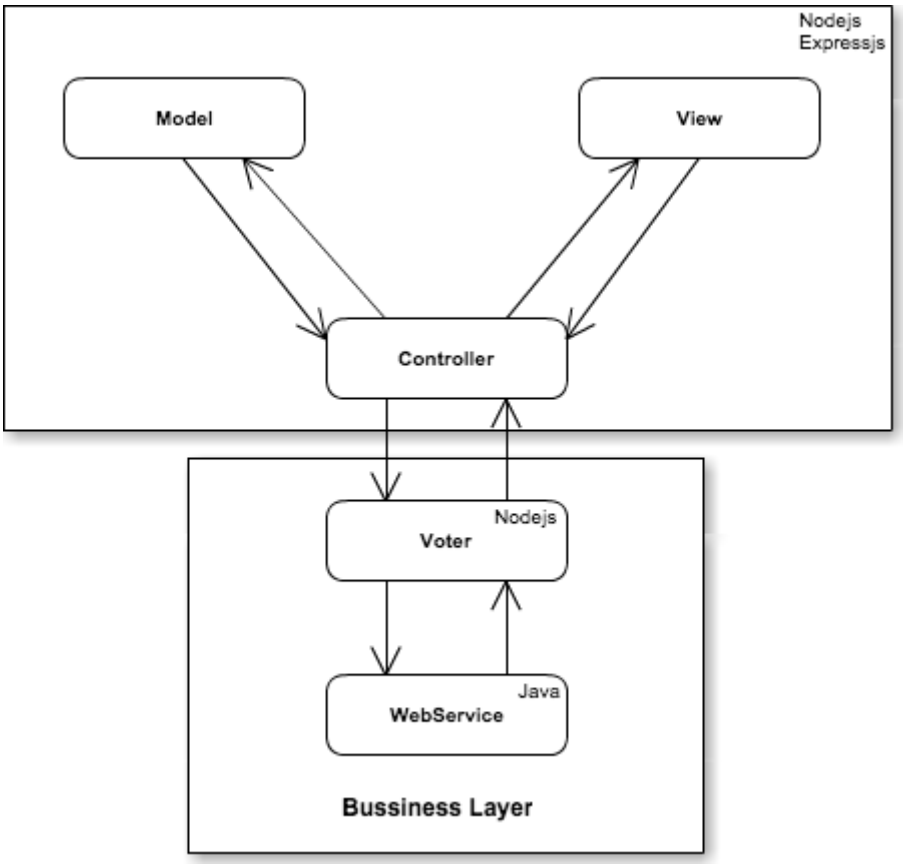

Figura 4. Diagrama de Arquitetura

3.2. Webservices

A calculadora de insulina foi desenvolvida em Java e recorre a JAXB para criar os *webservices* que permitem aceder aos vários métodos publicados. Este serviço implementa 3 métodos distintos a partir da sua especificação. Estes métodos são:

- **mealtime_insulin_dose** - calcula o número de unidades de insulina após uma refeição
- **background_insulin_dose** - calcula o número de unidades de insulina entre refeições
- **personal_insulin_sensitivity** - determina a sensibilidade (individual) a uma unidade de insulina

A cada grupo de trabalho foi atribuída uma máquina virtual para que os métodos possam ser publicados e acedidos pelos restantes grupos. Ao nosso grupo o endereço "*qcs11.dei.uc.pt*" e os *webservices* encontram-se disponíveis no endereço "http://qcs11.dei.uc.pt:8080/insulin/?wsdl", estando disponíveis os métodos supramencionados.

3.3. Votador

O *web server* (interface + votador) foi desenvolvido em Node.js (*javascript*) por ser uma ferramenta prática e fácil de integrar. Esta ferramenta foi também escolhida devido à própria natureza assíncrona da linguagem sendo *single-threaded* e capaz de superar sistemas *multi-threaded,* em determinadas condições, devido ao seu *Event-Loop*. Este *web server* contém duas componentes, uma que disponibiliza uma interface ao utilizador (web site) e a componente do votador que pode ser acedida através de uma API *REST*. Ao aceder a esta API o resultado será já o final depois de terem sido comparadas as N-versões.

Para colocarmos o *webserver* a funcionar com os clientes *WSDL* foi necessário instalar o módulo *node-soap* no Node.js que permite a ligação a este tipo de clientes (SOAP). Usualmente, quando se constrói um serviço em *Node.js*, utiliza-se *REST* para se fazer a comunicação entre vários serviços e o protocolo *SOAP* não será a melhor forma de comunicação nesta plataforma. Por este facto, o módulo instalado anteriormente possuía alguns erros de origem pelo que se teve de fazer alterações nessa biblioteca de forma a suportar a ligação aos clientes *WSDL* criados pelos diversos grupos. Este pequeno obstáculo foi superado mas existe ainda um problema quando os *webservices* possuem um *namespace* diferente do original

(“*http://server/*”). Considerando os *webservices* desenvolvidos pelos restantes grupos, não foram utilizados clientes cujo *namespace* difere de *http://server/*.

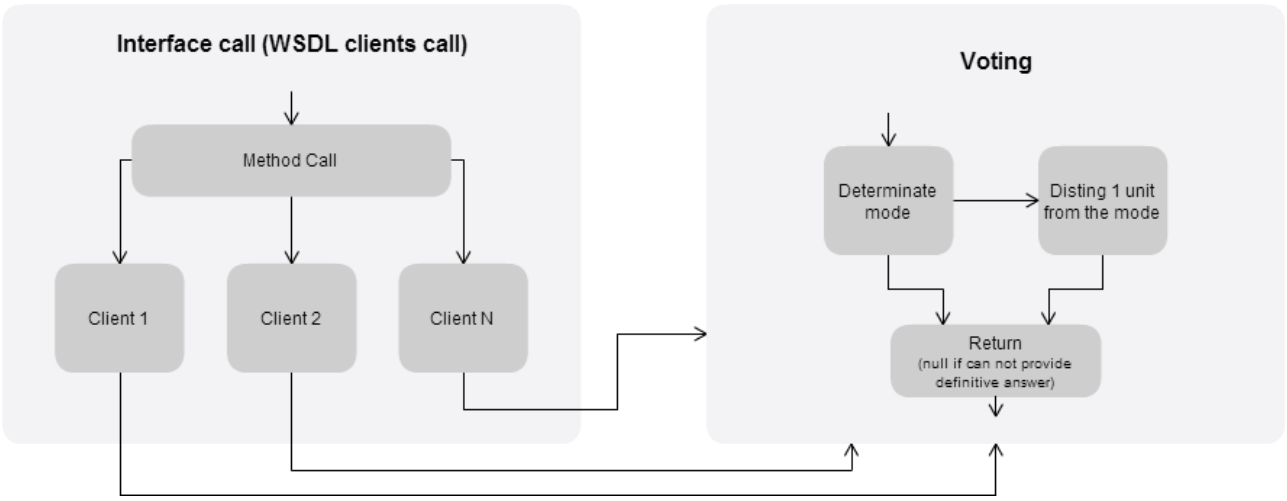

Figura 5. Esquema do Votador

Conceptualmente o Votador é um sistema simples que compara os resultados de execuções do mesmo procedimento em versões de *software* diferentes com o objetivo de estabelecer o resultado mais comum. Quando o utilizador pretende calcular a dose de insulina, o *web server* onde se encontra o Votador recebe a notificação e inicia o processo.

1. Para cada *webservice* é chamado o método remoto com os parâmetros do utilizador de forma assíncrona, com um *timeout* de 1 segundo.
2. Assim que os N clientes disponibilizam resposta (ou o seu *timeout* expira), os resultados são passados ao sistema de votação.
3. Tenta-se determinar a moda (valor mais frequente) das N respostas. Retorna este valor caso haja no mínimo 3 respostas para este valor. Se existirem vários valores para a moda (valores com a mesma frequência) retorna o valor mais baixo.
4. Caso não existam resultados suficientes (< 3), assume-se que valores que distam do valor mais frequente 1 unidade (± 1). Retornar caso sejam encontrados no mínimo 3 valores dentro deste critério. No caso de múltiplas

modas a comparação é feita a partir de todas as modas e é devolvido o valor que for mais baixo.

5. Caso o sistema de votação não consiga determinar uma resposta, retorna um valor nulo.

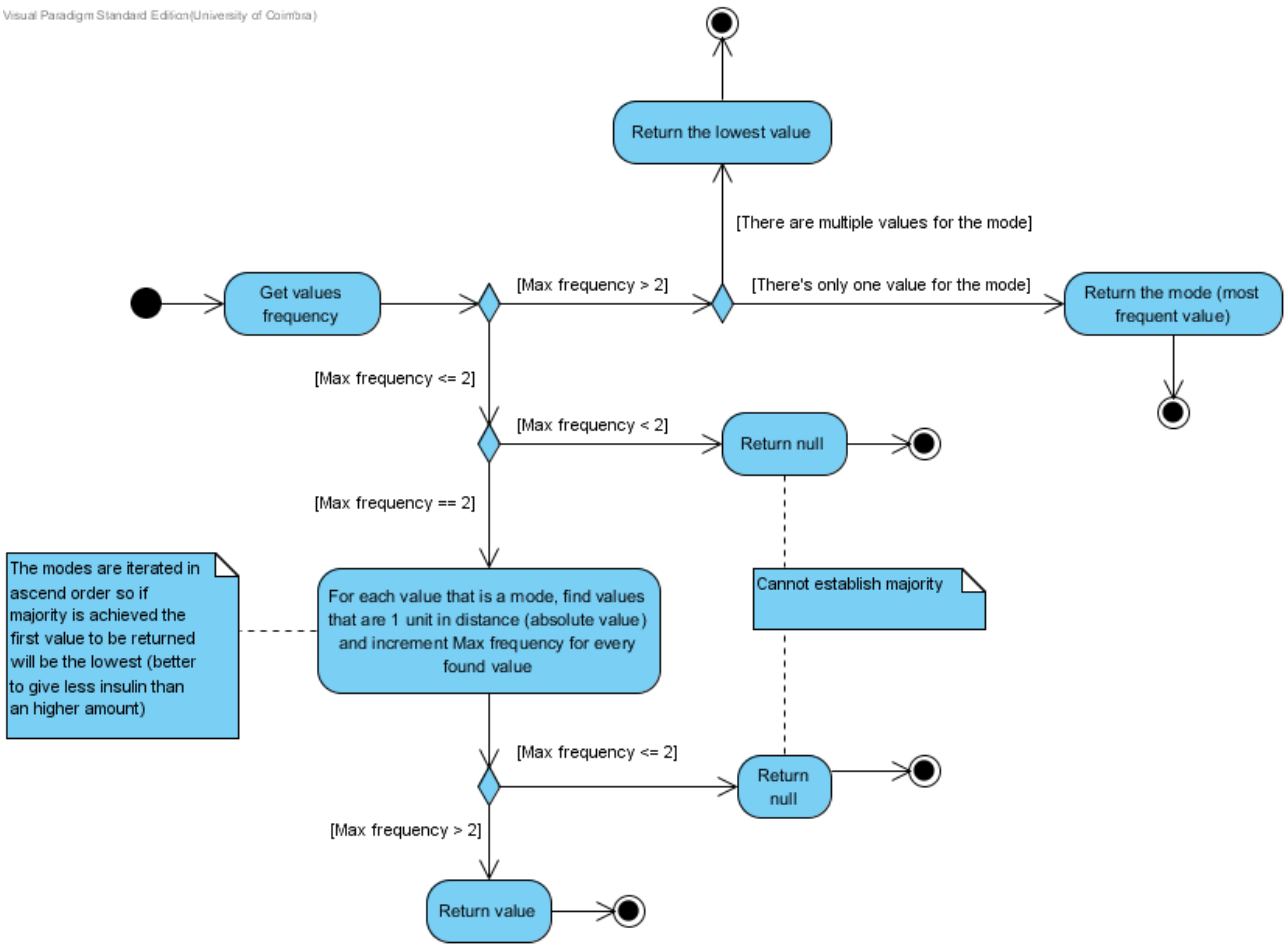

Figura 6. Funcionamento da função que determina o resultado maioritário

Javascript é uma linguagem assíncrona através de eventos e Node.js aproveita essa característica com a ajuda do *Event-Loop*. Desta forma, quando se executa um método moroso (como pedidos via internet), este método corre dentro do *Event-Loop* e assim que acabe de executar é lançado um evento que notifica que aquele pedido já concluiu. Desta forma, para executarmos os vários pedidos aos N *webservices* em simultâneo e sincronizar os seus resultados, usamos uma biblioteca bastante popular no mundo do *Javascript* denominada *async*. Esta biblioteca tem um método (*async.parallel*) que permite definir que funções executar e essas funções serão executadas em "paralelo" e só lança o evento quando todas as

funções terminarem. O método a executar nos diferentes clientes *WSDL* (*webservices*) são chamados em paralelo (na realidade não é em paralelo pois continua a existir apenas uma *thread*) e no final de todas as chamadas os resultados são passados ao método que avalia os resultados e determina o comum. Foi definido também um *timeout* de 3 segundos na chamada remota ao método. Caso este *timeout* seja ultrapassado, o pedido é automaticamente cancelado e retorna um valor nulo que é interpretado corretamente pela função de avaliação (ignorando o valor).

3.3.1. Interface web

O *website* está construído sobre as especificações fornecidas e encontra-se totalmente funcional. Ao entrar no *website* surge imediatamente um formulário que pode ser preenchido e por cima do mesmo encontra-se um pequeno menu de navegação que permite mudar para outro tipo de cálculo de insulina.

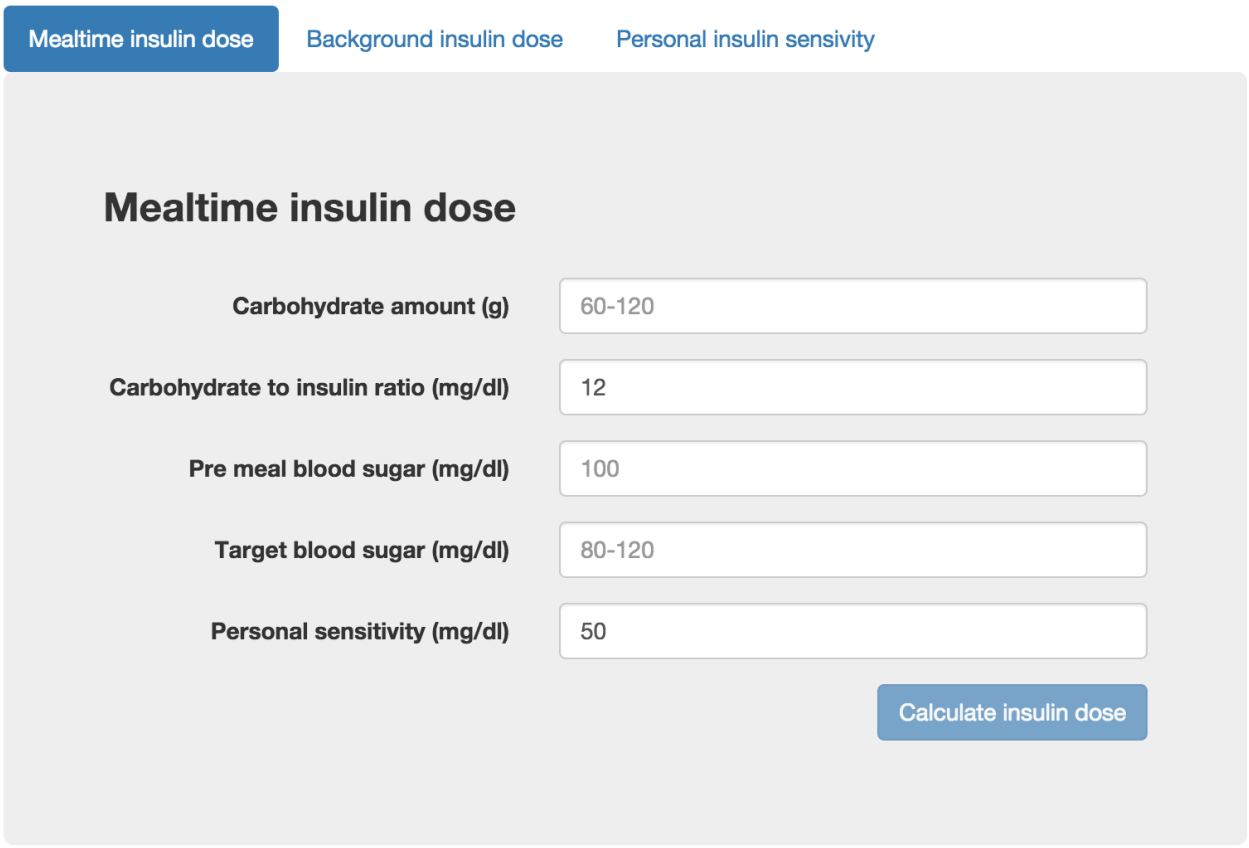

Figura 7. Formulário para o cálculo da dose de insulina à refeição

Quando o utilizador submete informação inválida num determinado campo é automaticamente apresentado o erro associado. Na imagem seguinte temos um exemplo em que foi inserido o valor 20 quando é permitido apenas inserir valores entre 60 e 120. Caso fossem inseridos caracteres alfanuméricos seria apresentado um erro relativamente ao campo permitir apenas caracteres numéricos. Desta forma o erro da parte do utilizador é minimizado, sendo confrontado com mensagens de erro ao executar incorretamente uma ação, sendo este *feedback* imediato uma funcionalidade importante numa aplicação utilizada por inúmeros utilizadores e grande maioria destes inexperientes.

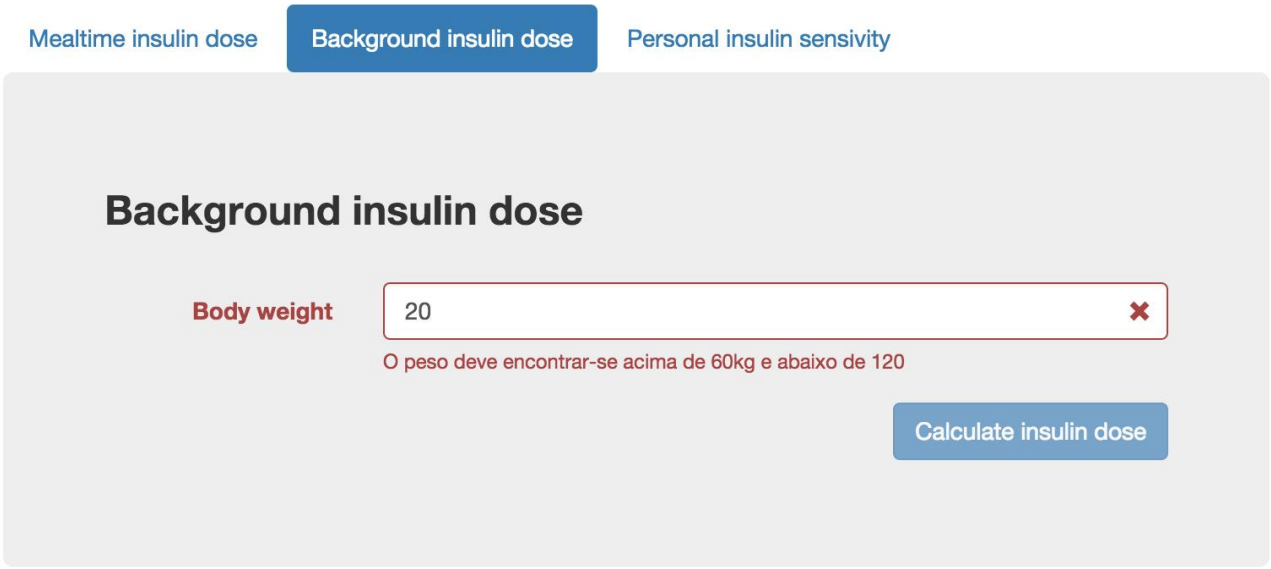

Figura 8. Mensagem de erro apresentada ao utilizador quando é inserida informação inválida

Existe também um botão que permite estender uma secção da interface, apresentando resultados detalhados do cálculo da dose de insulina. Esta secção é opcional e permite consultar os resultados individuais de cada versão do *webservice*. A imagem seguinte apresenta a secção detalhada, sendo possível ver o *webservice* que devolveu determinado resultado, bem como o tempo de resposta do mesmo. O sistema de cores permite facilmente distinguir os *webservices* que devolveram valores correctos (a verde), próximo da maioria (a amarelo) e valores errados e *timeouts* (a vermelho).

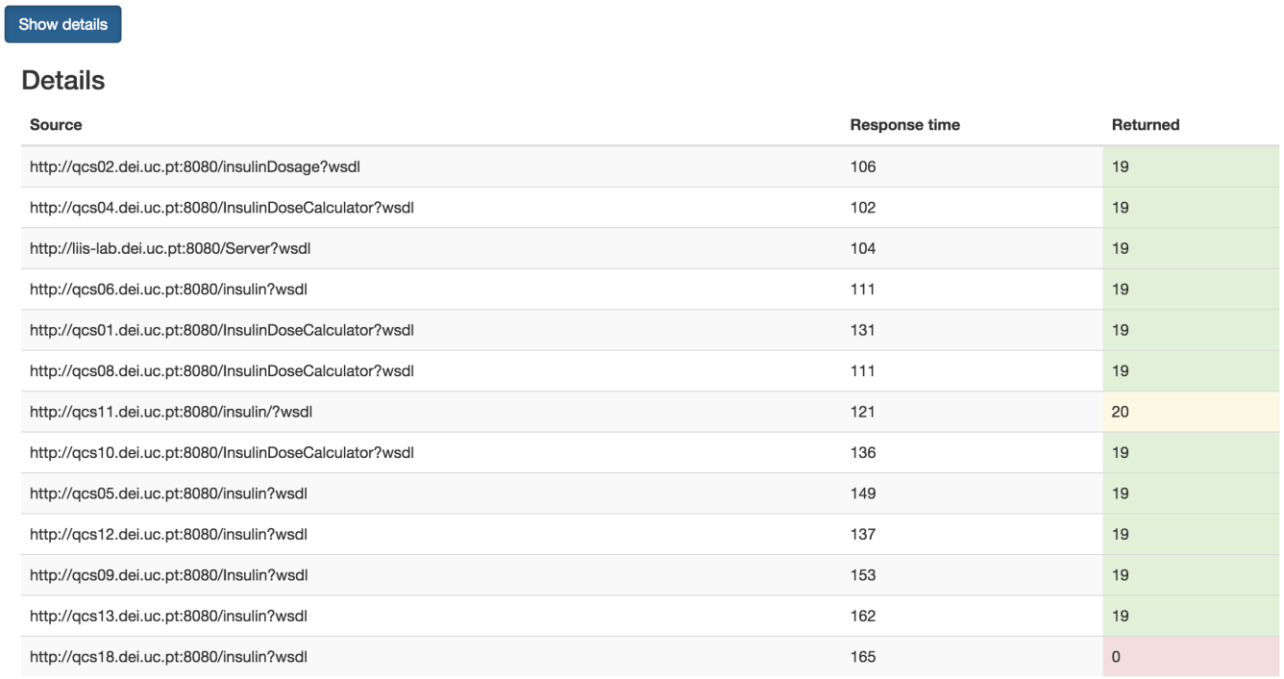

Figura 9. Detalhes do cálculo da dose de insulina

De acordo com a especificação NFR3, ao receber um resultado inválido do servidor é feita uma nova tentativa ao servidor. Perante a restrição de 4 segundos que o utilizador poderá esperar pelo resultado, são feitas no máximo 4 tentativas cada uma com um *timeout* de 1 segundo. Se no final destas 4 tentativas não receber ainda uma resposta válida, não é apresentado um resultado e surge um símbolo que indica a falha.

4. Validação

4.1. Testes unitários

Uma capacidade bastante importante de qualquer programador é realizar testes unitários. Testes unitários são testes que verificam partes individuais do código para garantir que cada módulo funciona corretamente. Para as várias unidades individuais (método ou parte de um método) que existam no programa, criam-se vários testes com o resultado que se espera obter com determinados *inputs*. Se o resultado obtido for igual ao esperado então o teste passa. Desta forma é

possível criar testes que validam os inúmeros módulos que um programa contém e automaticamente, aquando da compilação, é possível testar se alguma alteração afetou os testes, garantindo que se encontra ainda a funcionar como esperado. Poderão ser introduzidos mais testes unitários em qualquer momento do desenvolvimento do projeto.

4.1.1. Webservices

Com auxílio da ferramenta *JUnit* é possível testar automaticamente as várias componentes de um programa. Foram criados testes unitários com valores conhecidos disponibilizados pelo docente que nos permitiram validar o funcionamento correto dos métodos.

Meal time insulin dose

carbohydrate Amount	Carbohydrate To Insulin Ratio	Pre Meal Blood Sugar	Target Blood Sugar	Personal Sensitivity	Expected	Result
120	14	170	100	60	8	8
60	12	200	100	25	14	14
95	10	100	120	50	0	0

Background insulin dose

Body Weight	Expected	Result
79	22	22
62	17	17

Personal insulin sensitivity

physical Activity Level	physical Activity Samples	blood Sugar Drop Samples	Expected	Result
5	0,10	50,50	50	50
6	2,8	32,83	66	66
4	1,6,8,9	32,61,91,88	53	53

4.1.2. Votador

Para o votador criámos testes unitários com o objetivo de validar a função que calcula a maioria num conjunto de N valores. Foi decidido que a maioria corresponde a pelo menos uma repetição de um valor 3 vezes (haver um valor com uma moda de 3). Caso não haja 3 repetições mas apenas 2, consideramos haver repetição para valores que distam 1 unidade desta moda. Caso estas duas verificações falhem, o resultado é nulo (não foi possível determinar maioria).

Values	Expected	Result
5,2,7,4	null	null
5,null,6,3	null	null
5,5,null,null	null	null
5,5,7,7	null	null
1,2,3,4	null	null
3,3,3,4	3	3
5,5,6,6	5	5
6,6,5,5	5	5
5,5,5,6,6,6	5	5
3,3,4,4,5	3	3
0,18,0,18,19	18	18
22,0,22,22	22	22
-1,5,-1,0,5,5,6,-1,5	5	5

4.2. Model checking

Model checking consiste em testar exaustivamente e automaticamente um determinado modelo de um sistema de forma a verificar se o mesmo corresponde a determinadas especificações. Fazer este tipo verificação é necessária quando existe a necessidade de responder a certos requisitos de segurança, tais como a ausência de *deadlocks* ou de comportamentos que levam a um *crash*.

De maneira a resolver os problemas de forma algorítmica, é necessário que o modelo e as especificações do sistema a testar seja escritos numa linguagem matematicamente específica. Neste projeto a linguagem utilizada é a *PROMELA*, uma abreviação de *"PROcess MEta LAnguage"*. A *PROMELA* é uma linguagem baseada em C, contendo no entanto as suas especificidades. Além da linguagem existe também a necessidade de usar uma ferramenta que receba o modelo e a especificação e verifique se a especificação é assegurada pelo modelo. Neste projeto a ferramenta utilizada foi o *SPIN* de forma a validar o votador.

As seguintes funções são responsáveis pela chamada e simulação de resultados de cada *webservice,* respetivamente. É de notar que o *webservice* pode retornar um valor, neste caso entre 1 e 5, ou pode resultar em *timeout*.

```
//call webservices
for(i : 1 .. WS_CALLS) {
	run wsSimulator(i-1);
}

//simulate webservice result
proctype wsSimulator(int i) {
	int retval;
	if
	:: true -> select (retval : 1..5); wsResults[i] = retval;
	:: true -> time_out
	fi
}
```

Em seguida, depois de retornados os valores de cada *webservice*, o votador verifica o resultado maioritário. Desta forma, o algoritmo do votador, mencionado

na secção 3.3, foi implementado no *SPIN*. As seguintes funções simulam esse mesmo algorítmo.

```
//função que calcula frequencia de um dado valor
for(i : 0 .. WS_CALLS-1){
	if
	:: wsResults[i] != 0 -> arrayMode[wsResults[i]]++;
	:: else ->skip;
	fi
}


//funcao para encontrar o valor mais frequente
for(i : 0 .. SIZE-1) {

	if
	:: arrayMode[i] > max -> max = arrayMode[i]; index = i;
	:: else ->skip;
	fi
}
//valores que distam do valor mais frequente 1 unidade
if
:: max == 2 ->
	for(j : 1 .. MAXNUMBER) {
		if
		:: arrayMode[j] == 2 ->
			if
			:: (arrayMode[j-1] > 0 || arrayMode[j+1] > 0 ) ->
					max++; index = j; break;
			:: else -> skip;
			fi
		:: else ->skip;
		fi
	}
:: else -> skip;
fi

assert(max >= 0 && max <= MAXNUMBER);
```

Dado que o Votador foi desenvolvido em Node.js, não existe possibilidade de ocorrência de *deadlocks* devido ao funcionamento *single-thread* (apesar de poderem existir *race conditions* em ocasiões especiais quando a ordem dos *callbacks* é incorreta), pelo que o nosso principal objetivo com o *model checking* foi assegurar que o algoritmo do Votador apresenta os resultados esperados tal como foi enunciado na secção 4.1.2. Posto isto, através da verificação do modelo no SPIN e dos vários testes realizados, constatámos que o algoritmo é consistente e não apresenta erros. A seguinte figura demonstra o resultado da validação do modelo.

```
(Spin Version 6.4.3 -- 16 December 2014)
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +
State-vector 116 byte, depth reached 85, ••• errors: 0 •••
    21770 states, stored
     2625 states, matched
    24395 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
Stats on memory usage (in Megabytes):
    2.990       equivalent memory usage for states (stored*(State-vector + overhead))
    2.334       actual memory usage for states (compression: 78.07%)
                state-vector as stored = 84 byte + 28 byte overhead
  128.000       memory used for hash table (-w24)
    0.107       memory used for DFS stack (-m2000)
  130.353       total actual memory usage
unreached in proctype wsSimulator
        (0 of 14 states)
unreached in proctype voter
        (0 of 80 states)
pan: elapsed time 0.02 seconds
pan: rate   1088500 states/second
```

5. Conclusão

Com este trabalho conseguimos constatar a importância que programação N-versões tem em sistemas críticos. Através deste método, a probabilidade de existirem falhas de *software* idênticas em todas as versões é bastante reduzida, tornando assim o sistema mais viável. Porém, este método pode acarretar problemas como inconsistência de resultados, portanto é necessário garantir a validação do sistema. Numa aplicação como a que foi desenvolvida, através de programação em N-versões, podem existir simples erros de cálculo ou até mesmo de arredondamento que podem comprometer toda a confiabilidade do sistema e consequentemente criar danos no mundo real. Posto isto, é impreterível que se assegure a qualidade do sistema a desenvolver e que se proceda a uma correta validação do mesmo.