Instituto Tecnológico de Aeronáutica

Divisão de Ciência da Computação

Graduação em Engenharia de Computação

Prova Bimestral

Take Home Take Lab Test

Douglas Yamashita de Moura

Dr. Adilson Marques da Cunha

CES – 63 – Sistemas Embarcados

3º Ano Profissional – COMP07

São José dos Campos, 17 de Setembro de 2007

1. Objetivos

Esta prova do Primeiro Bimestre (*Take Home Take Lab Test – THTLT*) tem por objetivo a realização dos Tutoriais do *Rational Rose Real Time – RRRT,* também denominados *Warm-Up,* de números 1, 2, 3, 4, 5, 6 e 7.

A realização dos tutoriais tem como finalidade fazer com que os alunos das Disciplinas CES-63 aprendam as principais funcionalidades do *Rational Rose Real Time – RRRT e* possam utilizar os conhecimentos adquiridos no desenvolvimento do seu Protótipo de Unidade de Software de Computador – USC, o qual, por sua vez, deverá ser integrado no respectivo Componente de Software de Computador – CSC, que é constituinte do Item de Configuração de Software de Computador – ISC, resultando no Sistema de Software de Computador – SSC.

2. Conteúdo

Foram realizados os sete Tutoriais do *Rational Rose Real Time – RRRT (WarmUps)* e as principais funcionalidades praticadas em cada um deles estão relatadas nos itens a seguir.

2.1. Tutorial 1 (*WarmUp 1*) – *Hello World*

Neste primeiro tutorial, o aluno criou um modelo contendo um *capsule*, cujo comportamento era imprimir “Hello World!” na tela, aprendendo assim, a criar e salvar modelos no *Rational Rose Real Time – RRRT*; a criar um *capsule* na Visão Lógica do modelo; a adicionar um comportamento simples no *capsule*, através da criação e configuração de um estado na máquina de estados do mesmo, bem como de sua transição inicial.

Foram criados um *component* na Visão de Componentes e um *processor* na Visão de Implantação do modelo. Durante a realização dos passos realizados no tutorial, o aluno também teve noção de configuração da linguagem e do compilador a

serem utilizados, bem como o conceito de *Top Capsule*, e aprendeu como compilar, executar e debugar o modelo do RRRT.

Basicamente, foi criado um *capsule* contendo um estado denomidado *Active*. A transição inicial resultava neste estado e seu comportamento consistia em imprimir na tela a frase "*Hello World!*". A linguagem e o compilador utilizados foram, respectivamente, C++ e *NT40T.x86-VisualC++-6.0*.

2.2. Tutorial 2 (*WarmUp 2*) – *Passives Classes*

Neste segundo tutorial, o aluno criou um modelo contendo uma *passive class* contendo um método que possuía o código para imprimir uma mensagem de boas-vindas, aprendendo assim a criar um modelo utilizando-se de uma *passive class*; a adicionar um comportamento simples para a classe através da adição de um método; a compilar, executar e debugar este modelo; a alterar o código produzido pelo *Rational Rose Real Time* utilizando um editor de texto; e, a ressincronizar o modelo com o código alterado.

Assim como ocorreu no primeiro tutorial, foi criado um *component*, que continha a *passive class*, e um *processor*, que continha o *component*, sendo que neste modelo foi selecionado para exibição o *RTS view*, ao invés do *Model view* utilizado no primeiro tutorial. Após a compilação e execução do modelo, o aluno também aprendeu a modificar o código, a compilar e executar o modelo fora do RRRT e, a ressincronizar as alterações realizadas no modelo.

Basicamente, foi criada uma *passive class* com um método *main*, cuja função era imprimir uma saudação na tela para um terráqueo e, posteriormente, para um marciano. A linguagem e o compilador utilizados foram, respectivamente, C++ e *NT40T.x86-VisualC++-6.0*.

2.3. Tutorial 3 (*WarmUp 3*) – *Traffic Light*

Neste terceiro tutorial, o aluno criou uma aplicação simples de sinal de trânsito composta por um *capsule* com uma porta, cujo diagrama de estados continha três estados, e aprendeu a utilizar um diagrama de seqüência para construir tal modelo; a

criar uma classe simples de protocolo; e, a definir comportamentos, compostos por múltiplos estados e transições.

Inicialmente, foi criado um *capsule* e foi adicionada uma porta final a ele. Em seguida, foi configurado o comportamento deste *capsule* utilizando-se dos sinais do protocolo como *triggers*. Novamente, foram criados um *component* e um *processor*, e o modelo foi compilado, executado e debugado, sendo que durante esta última etapa, foram utilizados *probes* a fim de simulação de envio de sinais através das portas.

Basicamente, foi criado um *capsule* para modelar o funcionamento de um sinal de trânsito, que podia receber os sinais de controle *green, yellow* e *red* do protocolo. Tais sinais acarretavam em mudanças de estado, sendo que o primeiro alterava o sistema do estado Vermelho para Verde, o segundo do estado Verde para Amarelo e, o terceiro, do estado Amarelo para Vermelho. A linguagem e o compilador utilizados foram, respectivamente, C++ e *NT40T.x86-VisualC++-6.0.*

2.4. Tutorial 4 (*WarmUp 4*) – *Electronic Lock*

Neste quarto tutorial, o aluno modelou o comportamento de um cadeado eletrônico, utilizando-se de uma *passive class* com uma máquina de estados que provia tal comportamento desejado. Aprendeu a adicionar uma máquina de estados a uma *passive class*, a criar um método *trigger* para uma máquina de estados de uma *passive class*, o qual eram utilizados para as mudanças de estado, e a adicionar código a estas transições.

A configuração das ações e dos *triggers* fez com que fosse necessário a criação de uma operação para uma *passive class*, sendo que esta operação recebia um caractere como parâmetro e retornava um *trigger*.

Basicamente, foi criada uma classe que modelava o cadeado eletrônico, o qual só seria destravado se o usuário aperta-se as teclas 1 e 2 em seqüência. Qualquer outra seqüência utilizada pelo usuário é inválida e não alterará o estado de travado do cadeado. Caso o usuário aperte a tecla “?”, aparecerá uma ajuda na tela para o usuário.

Também foi criado um método na *main* que ficava em *loop* e recebia os caracteres inseridos pelo usuário e os retransmitia para a máquina de estados como *trigger*. A

linguagem e o compilador utilizados foram, respectivamente, C++ e *NT40T.x86-VisualC++-6.0.*

2.5. Tutorial 5 (*WarmUp 5*) – Battleship

Neste quinto tutorial, o aluno criou um modelo simples de simulação composto por navio de guerra que enviava sinails sonares pelo oceano em intervalos regulares e que detectava e mostrava os sinals retornados. Com esse modelo, aprendeu-se a criar um modelo utilizando-se de elementos já existentes, a configurar o comportamento de uma *capsule* com um timer, utilizando o *Timing Services* da biblioteca de serviços, a exibir mensagens de texto provenientes de uma *capsule*, utilizando o *Log Service* da biblioteca de serviços, e a compilar, executar, debugar e verificar o modelo usando diagrama de seqüência gerado durante o tempo de execução.

Basicamente, o modelo contém duas *capsules*, *Ocean* e Battleship, que atuam como *capsule role* dentro de uma *top capsule*, que seria a principal, denominada *World*. A *capsule Ocean* funciona como se fosse um submarino e pode estar ou não ao alcance do sonar da *capsule Battleship*. Quando está ao alcance, recebe o sinal e envia outro respondendo. Quando não está ao alcance, não tem nenhuma ação. A mudança de estados desta *capsule* é controlada por um *timer*¸o qual envia um *trigger timeout* no instante em que o tempo expira. Já a *capsule Battleship* possuía somente um estado e enviava sinais a cada segundo.

Para verificar o funcionamento do modelo, foi criado um diagrama de seqüência do mesmo, utilizando o *Trace* dos sinais. A linguagem e o compilador utilizados foram, respectivamente, C++ e *NT40T.x86-VisualC++-6.0.*

2.6. Tutorial 6 (*WarmUp 6*) – *Traffic System*

Neste sexto tutorial, o aluno criou uma aplicação de controle de tráfego baseanda em duas estradas que se interceptam (estradas que norte/sul e leste/oeste). Com isso, aprendeu a criar um novo modelo utilizando um modelo e elementos já existentes (no caso, utilizou-se o modelo já implementado do Tutorial 3), a criar uma nova *capsule class* e definir sua estrutura e comportamento, a criar uma *containing*

capsule, a adicionar estrutura para a *capsule* através da adição de *capsule roles*, e a compilar, executar e debugar este novo modelo.

Para controlar os semáforos, foi criada uma *capsule Controller* cuja máquina de estados atendia ao diagrama de seqüência exibido no tutorial. Sua estrutura foi definida através das portas, protocolos e *roles*, bem como com a utilização de serviços de tempo para determinar os instantes em que deveria ser alterados os estados. Este controlador funcionou também como uma instância da *Top Capsule Intersection*, a qual especificava que os semáforos opostos deveriam sempre estar no mesmo estado.

Basicamente, foram utilizadas as *capsules* acima mencionadas, e foi criado um *component* e um *processor* para se poder compilar, executar e debugar o modelo. O funcionamento do modelo foi verificado através do diagrama de seqüência criado do mesmo. A linguagem e o compilador utilizados foram, respectivamente, C++ e *NT40T.x86-VisualC++-6.0*.

2.7. Tutorial 7 (*WarmUp 7*) – *Traffic System*

Neste sétimo tutorial, o aluno criou duas aplicações simples do tipo cliente/servidor. Com isso, aprender a criar um modelo com *capsules* que são *incarnated* (dinamicamente criadas em tempo de execução), a criar *capsules* que são *imported* (dinamicamente movidas em tempo de execução), e a compilar, executar e debugar modelos que se utilizam dessas características.

O modelo implementado neste tutorial exemplificava uma conexão simples de um cliente com um servidor. Nesta conexão, o cliente enviava um pedido para o servidor em tempos regulares, sendo que ia para um estado de espera ao fazer isso. Por outro lado, o servidor ficava esperando por requisições do cliente, e, ao recebê-las, ia para um estado em que processava e enviava a resposta de volta ao cliente, retornando ao estado inicial logo após.

Basicamente, a *capsule role* foi criada como *optional*, o que faz com que a mesma não esteja ativa no início do modelo. Por outro lado, na *top capsule*, foi configurado que esta também fosse *optional*, mas que fosse ainda dinamicamente criada (*incarnated*). Dessa forma, simula-se uma situação em que o servidor estivesse sempre ligado e que o cliente pudesse se conectar a qualquer momento a ele, sem necessitar os dois estarem conectados no início da simulação. Em seguida, a mesma

role foi configurada para funcionar como um *plugin*, fazendo com que vários clientes pudessem se conectar ao servidor ao longo do tempo, e houve a necessidade de se criar uma nova instância de cliente para que a mesma pudesse se conectar dinamicamente *plugin* e se comunicar com o servidor.

O funcionamento do modelo foi comprovado através da utilização dos *traces* dos sinais e do *debug.* A linguagem e o compilador utilizados foram, respectivamente, C++ e *NT40T.x86-VisualC++-6.0.*

2.8. Síntese das Aplicações

Com a realização dos Tutoriais 1, 2, 3, 4, 5, 6 e 7 do *Rational Rose Real Time*, foram aprendidas as funcionalidades básicas do software, as quais serão aplicdas no desenvolvimento do protótipo de Unidade de Software de Computador – USC, no Componente de Software de Computador – CSC e no Item de Configuração de Software de Computador – ICSC.

Na USC Sistema Elétrico do VANT, muito provavelmente serão necessárias classes e *capsules* que visem atender aos requisitos especificados para a mesma, de modo a poder controlar e gerenciar as informações trocadas com as outras USCs.

No CSC Suporte do VANT, serão necessárias *capsules* que definam o comportamento de cada USC, ou seja, definirão as máquinas de estados que visam a coordenação e gerenciamento das ações a serem realizadas pelas USCs, tais como o envio de mensagens da USC Sistema Elétrico para a USC Central de Alarme no caso de algum corte de energia.

No ICSC VANT, todas as funcionalidades aprendidas serão utilizadas, pois este será responsável pelo gerenciamento e coordenação de todos CSCs constituintes, bem como da interação entre o próprio ICSC com os outros existentes no SSC. Desse modo, espera-se uma complexidade muito maior para esse nível do que comparado a uma simples USC.

3. Conclusão

A realização desta Prova Bimestral (*Take Home Take Lab Test – THTLT*) mostrou-se muito importante para o aprendizado das principais funcionalidades do RRRT, as quais serão praticadas em cada nível de integração do projeto, desde o desenvolvimento inicial da USC até a integração final do projeto VANT-EC-SAME.

Após a realização dos tutoriais, ficou evidente que o software *Rational Rose Real Time* facilita bastante todo o trabalho de desenvolvimento de um software embarcado, tornando todo o processo mais eficaz e eficiente.

Com o conhecimento adquirido, os alunos poderão implementar mais facilmente suas respectivas USCs e, posteriormente, as integrações em níveis mais altos, e farão essas integrações utilizando-se de ferramentas poderosas e de reconhecimento mundial.