Métodos Formais em Engenharia de Software

Mestrado Integrado em Engenharia Informática e Computação

Practical work on VDM++

Goals

The goal of this practical work is to develop, test and document an executable formal model of a high integrity software system in VDM++ using the Overture tool or the VDMTools. At the end of the work, students should have acquired the ability of formally modeling software systems in VDM++, and of demonstrating the consistency of the model. The work is to be performed by groups of two to three students attending the same class (turma). The groups will be defined and the themes assigned (different themes for different groups in the same class) in the practical classes of the week of 28 November to 2 December 2016..

Deliverables

The project should be organized in a way similar to the Vending Machine example (see the MFES page in Moodle). The practical work should be concluded and the deliverables (final report, VDM++ project files, UML project files) submitted in a .zip file through Moodle until **8am January 4**. It may be applied a penalty of 2 (points out of 20) for each day late. The presentations will be scheduled for the first week of January. Presentation is mandatory for **ALL** students. **If you miss presentation you will get zero points out of twenty.**

Report Structure

You should incorporate the developed VDM++ classes in a single report (to be submitted in PDF format), written in English or Portuguese, covering the following items (which are also the assessment items):

1. Front page with authors, date and context (FEUP, MIEIC, MFES) [1%]
2. Informal system description and list of requirements [10%]
    a. Requirements should include any relevant constraints (regarding safety, etc.).
    b. Each requirement should have an identifier.
    c. You may have optional requirements.
3. Visual UML model [5%]
    a. A use case model, describing the system actors and use cases, with a short description of each major use case.
    b. One or more class diagram(s), describing the structure of the VDM++ model, with a short description of each class, plus any other relevant explanations.
4. Formal VDM++ model [50%]
    a. VDM++ classes, properly commented.
    b. Needed data types (e.g., String, Date, etc.) should be modeled with types, values and functions.
    c. Domain entities should be modeled with classes, instance variables and operations. You are expected to make adequate usage of the VDM++ types (sets, sequences, maps, etc.) and create a model at a high level of abstraction.
    d. The model should contain adequate contracts, i.e., invariants, preconditions, and postconditions. Postconditions need only be defined in cases where they are

significantly different from the operation or function body (e.g., the postcondition of a sqrt(x) operation, which simply states that x = RESULT * RESULT, should be significantly different than the body); for learning purposes, you should define postconditions for at least two operations.

e. During the development of the project, if you foresee that the size of the VDM++ model will be less than 5 pages (or 7.5 pages in case of groups of 3 students) or more than 10 pages (or 15 pages in case of groups of 3 students), you should contact your teacher to possibly adjust the scope of the system or the modeling approach being followed.

5. Model validation (i.e., testing) [20%]
    a. VDM++ test classes, containing adequate and thorough test cases defined by means of operations or traces.
    b. Evidences of test results (passed/failed) and test coverage. It is sufficient to present the system classes mentioned in 4 painted with coverage information. Ideally, 100% coverage should be achieved. Optionally, you may include figures of examples exercised in the test cases.
    c. You should include requirements traceability relationship between test cases and requirements. Ideally, 100% requirements coverage should be achieved. It is sufficient to indicate in comments the requirements that are exercised by each test.
6. Model verification (i.e., consistency analysis) [5%]
    a. An example of domain verification, i.e., a proof sketch that a precondition of an operator, function or operation is not violated. You should present the proof obligation generated by the tool and your proof sketch.
    b. An example of invariant verification, i.e., a proof sketch that the body of an operation preserves invariants. You should present the proof obligation generated by the tool and your proof sketch.
7. Code generation [5%]
    a. You should try to generate Java code from the VDM++ model and try to execute or test the generated code. Here you should describe the steps followed and results achieved.
8. Conclusions [3%]
    a. Results achieved.
    b. Things that could be improved.
    c. Division of effort and contributions between team members.
9. References [1%]

FAQs

Why are my post-conditions similar to the function's body?

Au contraire, you should actually be wondering: "why is the body function a repetition of the post-condition"? In most cases such scenario is perfectly natural; for example, if the postcondition of a function is "the return value is the product of two arguments", then probable the function's body will use the same arithmetic operations to achieve the end result. However, that's not always the case. Take a function that return the square root of a number. The post-condition could be easily expressed

as “the result, whatever it is, when squared, is equal to the input”. The body would be much more complex, since it needs to actually “compute” the root. The same thing goes for multiplication; one could say that multiplying A by B is summing B times the number A.

So is one allowed to repeat the body in the post-condition?
For sufficiently complex functions, the repetition only occurs because one tend to first build the function body and then the postcondition. The latter would then assume an “algorithmic” expression, instead of a “declarative” assertion. The rule of thumb is to invert the rationale: first one should build the intended post-condition and only then the specific algorithmic implementation. The philosophy is akin to the difference between TFD (Test-First Development) and TLD (Test-Last Development).

So... one should always design the contracts first?
Absolutely! And before the contracts, one should make the types as strong as possible (they are the de facto first line of formalization). Sufficiently strong types can easily replace a number of invariants, pre and postconditions.

What makes a good pre/postcondition?
Pre-conditions restrict the scenarios where an operation can take place. Hence, a good pre-condition will be one that is as weak as possible to allow the correct functioning of the operation in the widest possible situations, but not weaker to the point where it accepts invalid status. This is commonly known as the weakest precondition. Example: given a function sqrt: Double → Double, a possible precondition would be input > 1. Although valid, it’s not the weakest one, since it’s discarding 0 and 1 as possible (and perfectly acceptable) inputs. An invalid (though weaker) pre-condition would accept negative numbers as an input, something whose square root cannot be expressed in the domain of Doubles.
Post-conditions specify the ending scenario after an operation took place. Hence, a good post-condition will be one that is as strong as possible to exactly define what the operation will end up doing in the widest possible situations, but not stronger to the point where perfectably valid results would not be accepted. This is commonly known as the strongest postcondition. Example: given a function primesUntil: Nat → Seq of Nat, a possible post-condition would be that the result should be a subset of all odd numbers up until N (RESULT in { x | x: nat & x < N and x mod 2 = 1}). Although all prime numbers are odd, some odd numbers are not prime. Hence, the post-condition should be made stronger.

Temas

1- LinkedIn

Neste projeto pretende-se modelar a rede social Linkedin. Nesta rede é possível, entre outras coisas, determinar a distância entre duas pessoas, colocar CVs, procurar pessoas, calcular os contactos comuns entre duas pessoas, determinar a pessoa com mais contactos, calcular a distância média entre pessoas na rede, etc.

2- Quaridor

A ideia do jogo é levar o seu peão até o lado oposto do tabuleiro. Seu oponente deve fazer o mesmo, obviamente. E quem chegar do outro lado primeiro, ganha.
Ocorre, porém, que você tem algumas vedações/cercas que servirão para impedir o avanço de seu oponente, que fará o mesmo.
http://www.jogos.antigos.nom.br/quoridor.asp

3- Paciência de pirâmide

Retire as cartas da pirâmide fazendo combinações de cartas com um valor combinado de 13 pontos.
http://www.cadajogo.com.br/jogo/pyramid_solitaire_1.html

4- Xo Dou Qi

O Xo Dou Qi (ou Dòu Shòu Qí, no pinyin mandarim) surgiu na china em data incerta. É um jogo que utiliza animais esculpidos como peças do jogo, num tabuleiro dividido em várias casas com respetivas inscrições. No Xo Dou Qi é preciso saber correr riscos, avançar na hora certa e recuar quando for preciso.
http://pt.wikipedia.org/wiki/Xo_Dou_Qi

5- Combate

Combate foi um jogo de tabuleiro lançado pela Estrela em 1974 com o nome de Front, posteriormente substituído por Combate. Nos países de língua inglesa é conhecido como Stratego1 .
É um jogo praticado por duas pessoas em que o objetivo é capturar a bandeira adversária, que atualmente foi substituída por um prisioneiro.
http://pt.wikipedia.org/wiki/Combate_(jogo)

6- Jogo Marel

Marel é uma variante do jogo da velha, embora com mais opções. É da família dos jogos conhecidos como "Merels", da qual descende também o jogo de "trilha" ou "moinho", que Nigel Pennick, no seu "Jogos dos Deuses" chama de "Nine Men's Morris". Em todos, a finalidade é a mesma, ou seja, colocar três peças em linha. Num tabuleiro com linhas horizontais, verticais e diagonais, os jogadores vão colocando uma peça por vez. Em algumas variantes, o número de peças é de 9, como, obviamente, no caso do "Nine Men´s Morris". Em outras, são 3 ou 5 peças. Após esta fase, os jogadores podem movimentar as suas peças, uma por vez, até uma intersecção de linhas. Ganha quem colocar as suas peças em linha, seja na horizontal, vertical ou diagonal.
http://en.wikipedia.org/wiki/Nine_Men%27s_Morris

7- Jogo da batalha naval

O jogo é jogado em quatro tabuleiros, duas para cada jogador. Os tabuleiros são tipicamente quadrados - geralmente 10 × 10 - e as células individuais no tabuleiro são identificados por letra e número. Num dos tabuleiros, o jogador organiza navios e regista os tiros do adversário. No outro tabuleiro, o jogador regista os seus próprios tiros. Antes do jogo começar, cada jogador dispões secretamente seus navios no seu tabuleiro principal. Cada navio ocupa um número de quadrados consecutivos no tabuleiro, dispostos horizontalmente ou verticalmente. O número de quadrados em cada navio é determinado pelo tipo de utilização do navio. Os navios não podem se sobrepor (ou seja, apenas um navio pode ocupar qualquer quadrado dado na grade). Os tipos e números de navios permitidos são os mesmos para cada jogador. Estas podem variar em função das regras.
http://en.wikipedia.org/wiki/Battleship_(game)

8- Google Scholar

O Google Scholar é uma base de dados de referências bibliográficas disponível online. Aqui é possível pesquisar investigadores e seus co-autores. É ainda possível ver a lista de publicações científicas de cada investigador. Para cada publicação é ainda possível ver as publicações referidas, as relacionadas, etc. Podem obter mais informação em http://scholar.google.pt

9- Tetris

Tetris é um jogo eletrónico muito popular, lançado em junho de 1984. O jogo consiste em empilhar tetraminós que descem a tela de forma que completem linhas horizontais.] Quando uma linha se forma, ela se desintegra, as camadas superiores descem, e o jogador ganha pontos. Quando a pilha de peças chega ao topo da tela, a partida se encerra. Ver mais informação em http://pt.wikipedia.org/wiki/Testris.

10- Modelos de *features*

Modelos de *features* (ver https://en.wikipedia.org/wiki/Feature_model) são utilizados para representar variabilidade numa linha de produtos ou para representar as configurações possíveis de um sistema. Dado um modelo de *features*, uma configuração é uma seleção válida de *features* (isto é, um conjunto de *features* que obedece às restrições definidas no modelo de *features*). Construir em VDM++ um meta-modelo formal para representar modelos de *features*, criar incrementalmente modelos de *features*, verificar se uma configuração é válida (em relação a um modelo de *features*) e gerar todas as configurações válidas em relação a um modelo de *features*.

11- "Andante"

Modelar em VDM++ o comportamento das máquinas de venda automática de títulos de transporte "Andante" (ver http://www.linhandante.com/). Opcionalmente, modelar também o comportamento das máquinas de validação automática e criar casos de teste que simulam o ciclo de vida de um cartão.