Formal Methods in Software Engineering

Mestrado Integrado em Engenharia Informática e Computação (MIEIC)

2018/19

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).

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 **8 am January 7**. 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 (4th or 5th of January). The presentation is mandatory for **ALL** students. **If you miss the 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. Evidence 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 the 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 probably 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 returns 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 tends 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: one should build the intended post-condition first 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 perfectly valid results would not be accepted. This is commonly known as the strongest postcondition. Example: given a function *primes Until*: *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.

Themes

1. Kids2Kids (http://kidtokid.com/)
As your kids grow and learn, sell back their stuff to Kid to Kid and find kids clothing, gear, and toys that are exactly what they need for their next step. It's smart & easy.

- Classify products in different classes
- Get products from suppliers/clients
- Sell products
- Different stores around the world
- Reports with different metrics, etc.

2. Company with a distributed printing service

- Distributed printing queue depending on the type of document (black&white, color, A3, A4)
- Payment with balance
- Printers placed in different places/clients
- solve malfunctions reported / send employees to fix those
- Reports with several metrics, etc.

3. ShopAdvizor (http://www.trialpanel.com/)
The first collaborative community that interconnects people, retailers, and brands. Ratings & Reviews, sales boost and key shopper data.
Through the mobile app, users have the opportunity to find information about your products (ratings and reviews, composition, nutritional information …), and also to participate in various competitions and missions offered by your brands.

4. PerfectGym (https://www.perfectgym.com/)
PerfectGym is a powerful mix of club management solutions. Over 20 user-friendly and easy to implement features will let you manage, grow and optimize your business. You gain access control, advanced reports (club statistics, client activity), CRM database, newsletter/SMS module, invoice generation, calendar module for trainers and SPA and much more.

5. Glovo (https://glovoapp.com)
Glovo is the app that allows you to get the best products of your city in minutes. Tell us what you want and we will deliver it to you wherever you are.
The best products in your city? Like what? A charger, food, a smoothie, a hot meal from your favorite restaurant, your keys…
Just think of what you wish... Everything in your city, delivered in minutes.

6. Turo (https://turo.com/)
Turo is a car sharing marketplace where travelers can book any car they want, wherever they want it, from a community of local car owners.

7. Uber
Get a ride in minutes. Or become a driver and earn money on your schedule. Uber is finding you better ways to move, work, and succeed.

- operates in different countries
- several employees
- several different cars
- each ride has a price
- reports with several metrics, etc.

8. Rome2Rio (https://www.rome2rio.com/)
Discover how to get anywhere by plane, train, bus, ferry & car

9. GitHub (https://github.com/)
GitHub Inc. is a web-based hosting service for version control using Git.

10. Agenda Viral (https://www.viralagenda.com/)

- find events
- promote your events
- buy tickets
- reports with several metrics, etc.

11. Facebook (https://www.facebook.com/)

- several users registered
- each user has friends
- each user may define lists of friends with restricted access
- posts
- etc.

12. FIL - Centro de exposições de Lisboa (https://www.fil.pt/)

- schedule of the events
- visitors
- exhibitors
- cost
- profit
- report with several metrics, etc.