CSC3323 Software Verification Technology

Main Coursework 2017/18

Leo Freitas

October 2017

Objectives

The purpose of the tasks below is to ensure that you can:

- read and write modelling scenarios in VDM/Overture;
- make appropriate modelling decisions;
- understand the role of specification (i.e. invariants, pre, post, etc.);
- understand the role of proof obligations;
- prove in Isabelle simple properties of interest about models;

Support will be provided in practical classes and lectures.

This coursework is marked (40% overall mark)!

Submission details:
Submit your (zipped) Overture, Isabelle and other file(s) to NESS.

VDM modelling deadline (w11) Wed, 15th Nov 2017, 11:59pm.
Isabelle proving deadline (w15) Thr, 14th Dec 2017, 11:59pm.

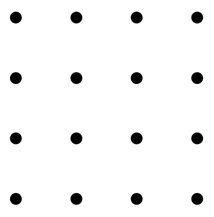
First deadline mark has low weight and serves to give you feedback on your VDM model. This will give you a chance to know what is wrong and how to fix it in time for the second (final) deadline.

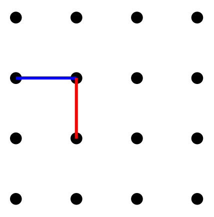
Submission files:
Each student must submit their own files (e.g. SurnameStdNo.ext). Overture files will contain your VDM model (e.g. Smith123.vdmsl); Isabelle files will have the VDM model translation (e.g. Smith123.thy) and proof obligation scripts. There is a file template on Blackboard for both file types. Please use them to avoid confusion. You are encouraged to add textual explanations as comments and/or literate programming.

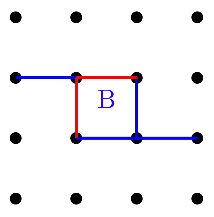
1 Problem Scenario

Description

Dots-and-boxes is a pencil-and-paper game for i players of different colours who take turns within the spaces in an $n \times n$ grid. We illustrate the play on a 4×4 grid with two (blue, red) players (see grids in the picture below). Players take turns by joining exactly two dots with their corresponding colour in either a horizontal or vertical (but not diagonal) move (see middle grid). The game play must be fair: no player can play twice in a row; and whoever plays the fourth side of a box owns it, regardless of the surrounding colours (see right grid). Every box made leads to a mandatory extra move. Deciding when to make a box is optional (i.e. players can avoid it) and is part of possible winning strategies. The player with the higher number of owned boxes at the end wins.


Initial grid


Players take turns



Blue player owns a box

The game requirements, move types, and winning strategies are described in detail at https://goo.gl/ahDQSk and a video at http://goo.gl/oFhLRf. You can also play it online at http://dotsandboxes.org. This game enjoys a number of interesting mathematical properties. For instance: there is a winning strategy for two players on grids with a “central box” (i.e. when n is even); the number of available moves is a function of the grid size and possible boxes; etc. The game is also useful to characterise (mathematically) interesting properties of interest, such as: who won?; what is the minimum/maximum number of moves possible to win?; what “types” of move are there?; etc. The game will be used it to ascertain your understanding of modelling and proof techniques.

2 Tasks

Description

Using Overture and Isabelle, your task is to:

- create the board layout,
- define players valid-moves
- encode the game-play itself as a state-based model with operations

This VDM model does not need to be executable, but it might be easier to debug if you can "run" the game. A suggested state and top-level algorithm is given in Box on the next page. You can adjust it as you see fit. To complete the model for the suggested algorithm you will need to define data types, auxiliary functions and operations. Keep your model tidy; proper indentation and succinct comments are useful.

Next, you will need to translate the model to Isabelle in order to discharge proof obligations about satisfiability (i.e. operations contracts are feasible) and sanity checks (i.e. the game behaves as expected) for your model in Isabelle.

Hint:
Decomposing the problem (as you see fit) and writing auxiliary functions for invariants and other properties help simplifying the solution by making it modular. Important issues to consider include (but are not limited to):

- types and invariants for board, position, player move, kinds of moves, etc;
- calculate the game stage (e.g. account for boxes, available moves, etc.);
- characterise moves so far, as well as those available to be taken;
- represent winning conditions and who won;
- specify the game state representation and invariants;
- write tests to convince yourself the model is suitable;
- prove satisfiability for operations involved;
- modelling decisions might affect how easy/hard proof work will be.

Aspects of the model to help you brainstorm solutions will be presented.

```
/**************************** Game state ****************************/
-- state records each of the players moves, who is playing, and all
-- remaining possible moves. The play algorithm ensures fairness
-- (i.e. players take turns, and making a box awards another move).
-- Initially p1 start, no moves are played all valid moves are possible.
state DBGame of
  p1: Moves
  p2: Moves
  p1plays: bool
  possible_moves: set of set of Move
  inv mk_DBGame(p1, p2, -, possible_moves) ==
    valid_moves(possible_moves)    -- only valid moves are possible
    and
    ....                           -- players don't share moves
    and
    ....                           -- played moves aren't available
  and
  ....                             -- other invariants of interest
  init s == s = mk_DBGame(...)
end

/****************** Fair play top-level algorithm ******************/
-- top-level algorithm for fair game play:
-- a) while there are moves left
--     1) choose a coordinate
--     2) calculate the move for coordinate
--     3) update state with chosen move
--     4) test whether move made box
--     5) flip player if so; clear box from state otherwise
-- b) tally and print results
play() ==
  ((while moves_left() do
    (dcl whop   : seq1 of char := (if p1plays then "P1" else "P2"),
         choice : Coordinate := silly_choose(),
              m : Move := move(choice);
           save(m);
           print("Player " ^ whop ^ " move: ");println(m);
           (if not player_made_box() then
             flip_player()
           else
             clear_box_made();
             println("Player " ^ whop ^ " closed a box!")  )
    ));  tally())
ext
  wr p1plays, p1, p2, possible_moves
post
  dunion possible_moves = {};
```

Task A (35/100): deadline (w11) Wed, 15th Nov 2017, 11:59pm.
Write the game as a VDM-SL model

Purpose:

Modelling decisions variety; documentation of design decisions.

Question:

Provide a model to the problem scenario. This ought to include:

- constant, type, and invariant declarations;
- description of kinds of moves possible (see video @ goo.gl/oFhLRf);
- state invariant, pre and postconditions for operations;

You may also find useful to be modular, and use:

- (auxiliary) functions and operations;
- boolean-valued functions for (reusable) invariants, etc.

Explain and justify your choices with comments to inform the reader of your design decisions. Use VDM/Overture to typeset your model. Models do not need to be executable.

Answers using value enumeration (e.g. manually writing all possible moves will be tedious and error prone and) will not be considered. They can be useful for debugging but must not be your final answer.

Hint:

Overture helps avoiding trivial mistakes and provide proof obligation statements to be proved in Isabelle (see Tasks C–D). It is useful to separate your model (and its invariants) per part.

1. Constants, types, auxiliary functions
2. Points of interest
3. Move types, and taken
4. pre/post conditions in auxiliary functions
5. State invariants
6. pre/post conditions in state operations
7. extended explicit operations are easier to follow
8. etc.

Task B (30/100): deadline (w11) Wed, 15th Nov 2017, 11:59pm.
Translate your VDM model to Isabelle

Purpose:

Understand interplay between VDM modelling and Isabelle proving.

Question:

Translate your model from Task A into Isabelle and check it using Isabelle's value commands and/or auxiliary lemma statements.

Convince yourself (and write comments about why) the model makes sense. Have you built the model right? Have you built the right model?

Hint:

- nitpick can help debug your model;
- sledgehammer or "by auto" can help proving conjectures;
- If a proof is "too complicated", try simplifying your model;
- Isabelle does not enforce (implicit type/state) invariant/pre/post checks;
- Be systematic: translate it as if you had "a program" do it for you.

Task C (20/100): deadline (w15) Thr, 14th Dec 2017, 11:59pm.

Discharge satisfiability proof obligations (POs) in Isabelle

Purpose:

Understand modelling consequences in practice (through proof)

Question:

Using Isabelle, state and prove: satisfiability proof obligations as given by Overture for majority of operations defined (i.e. moves_left, save, etc);

Revise and update your VDM model and Isabelle translation depending on results from (failed) proofs;

The more you prove (i.e. auxiliary functions and operations), errors can be compensated (i.e. if your model have 8 operations and you prove 6 POs; you can afford some mistakes in at least 2, since 4 correct proofs would suffice).

Hint:

- Look at proof exercises from lecturers;
- Write proof scripts, even if partial;
- Write a proof plan as comments if you fail to do it using Isabelle;
- If the model datatypes/invariants are chosen carefully, proofs commands like "by auto" or "by simp" should suffice after definition expansion.

Notes:

The satisfiability proof obligation of an operation *Op* under state *St* is:

```
forall input: TypeI, before: State & pre_Op(input, before) =>
   (exists output: TypeO, after: State &
       post_Op(input, output, before, after))
```

That is, given any input and before state, if the operation precondition holds, then find me witnesses for the output and after state, such that the operation postcondition holds. Operations without inputs or outputs can be declared similarly without the parameters.

Overture PO generator (POG) produces different versions of the satisfiability PO, depending on the kind of VDM declaration sed (e.g. implicit, explicit, extended). In essence the POG expand/simplifies definitions, as well as take advantage of explicit specification statements as witnesses to existential quantifiers. In doubt, use the general template above when translating to Isabelle.

Task D (10/100): deadline (w15) Thr, 14th Dec 2017, 11:59pm.
State and prove sanity checks in Isabelle

Purpose:

Understand the difference between verification and validation

Question:

Having built a convincing model and proved its satisfiable, it is important to ensure that the model you have does what you want. That is the difference between verification (e.g. build the model right: it is satisfiable) and validation (e.g. build the right model: it behaves as expected).

Declare and prove in Isabelle at least two sanity checks of your choice. These can be (but are not limited to):

1. there can only be one winner;
2. available moves are within maximum board positions;
3. number of moves is fair (i.e. one move per player per round), etc.

As with satisfiability, if you prove more than 2, errors can be compensated.

Hint:

You can use Isabelle's value command with concrete values on the chosen sanity check above, or use nitpick to find mistakes or misunderstandings quickly. It is similar to debugging in Overture.

Task E (5/100): deadline (w15) Thr, 14th Dec 2017, 11:59pm.
Reflect on your learning experience

Purpose:

Think about what you learned; reflect on outcomes; give feedback.

Question:

Looking back at your design decisions, how did they affect the proofs involved? If anything, what would you have done differently and why?

Reflect and write on the modelling and proving Tasks A—D. What were your best/worst design decisions and why? How would you do it differently?

Write a brief personal account of the material you found hard to understand and how you overcame problems. What did you find difficult / interesting / worthwhile / fun in this course?

Hint:

Feedback on the material and style of presentation will also be much appreciated. This is a real chance to both say something technical and improve the course for future years!

Vacuous answers like "nothing was difficult" will only attract non-zero marks if other parts of the coursework were not superbly done ☺!

3 Extras — useful for Tasks C and D

Natural Deduction Rules

Operator	Introduction	Elimination
$\wedge$	$\dfrac{A \quad B}{A \wedge B}$	$\dfrac{A \wedge B}{A}$ $\qquad$ $\dfrac{A \wedge B}{B}$
$\vee$	$\dfrac{A}{A \vee B}$ $\qquad$ $\dfrac{B}{A \vee B}$	$\dfrac{A \vee B \quad \begin{array}{c}\lceil A \rceil \\ \vdots \\ R\end{array} \quad \begin{array}{c}\lceil B \rceil \\ \vdots \\ R\end{array}}{R}$
$\longrightarrow$	$\dfrac{\begin{array}{c}\lceil A \rceil \\ B\end{array}}{A \longrightarrow B}$	$\dfrac{A \longrightarrow B \quad A}{B}$
$\longleftrightarrow$	$\dfrac{A \longrightarrow B \quad B \longrightarrow A}{A \longleftrightarrow B}$	$\dfrac{A \longleftrightarrow B}{A \longrightarrow B}$ $\qquad$ $\dfrac{A \longleftrightarrow B}{B \longrightarrow A}$
$\neg$	$\dfrac{\begin{array}{c}\lceil A \rceil \\ \textbf{false}\end{array}}{\neg A}$	$\dfrac{A \quad \neg A}{\textbf{false}}$
false	$\dfrac{A \quad \neg A}{\textbf{false}}$	$\dfrac{\begin{array}{c}\lceil A \rceil \\ \\ \textbf{false}\end{array}}{\neg A}$ $\qquad$ $\dfrac{\begin{array}{c}\lceil \neg A \rceil \\ \\ \textbf{false}\end{array}}{A}$
true	$\dfrac{}{\textbf{true}}$	N/A

Isabelle Natural Deduction Rules Correspondence

Operator	Introduction	Elimination
$\wedge$	conjI	conjE, conjunct1, conjunct2
$\vee$	disjI1, disjI2	disjE
$\longrightarrow$	impI	impE, mp
$\longleftrightarrow$	iffI	iffE
$\neg$	notI	notE
True	TrueI	TrueE
False	N/A	ccontr, FalseE

Isabelle's meta-logic

- Isabelle's "$\Longrightarrow$" represents meta-implication (i.e. the "bar" on first Table);
- Use find_theorems on your goal to find out other useful rules.