Bridge Requirements (Spec): EECS4312

JSO,EECS

November 6, 2018

©**This document is not for public distribution**. This document may only be used by EECS4312 students registered at York University. By downloading this document from the department, registered York students agree to keep this document private for their personal use, and may not communicate it to anyone else.

Students may work in pairs provided both partners contribute substantially to the Lab, but must otherwise obey York regulations on academic honesty requiring that students not plagiarize the work of others or submit the work of others.

Contents

List of Figures

List of Tables

1 Introduction

For the Bridge problem, see Fig. 1. You elicit the requirements and specify the Computer Controller (SUD) as a complete and disjoint function table which you use to validate the requirements with invariants and use cases.

You may be required in the submission to write a Requirements Document. Even if you are not asked to write a Requirements Document, you might want to use this example to practice the art of writing a complete requirements document (atomic E and R-Descriptions, list of monitored/controlled variables, function table specification and validation of the function table).

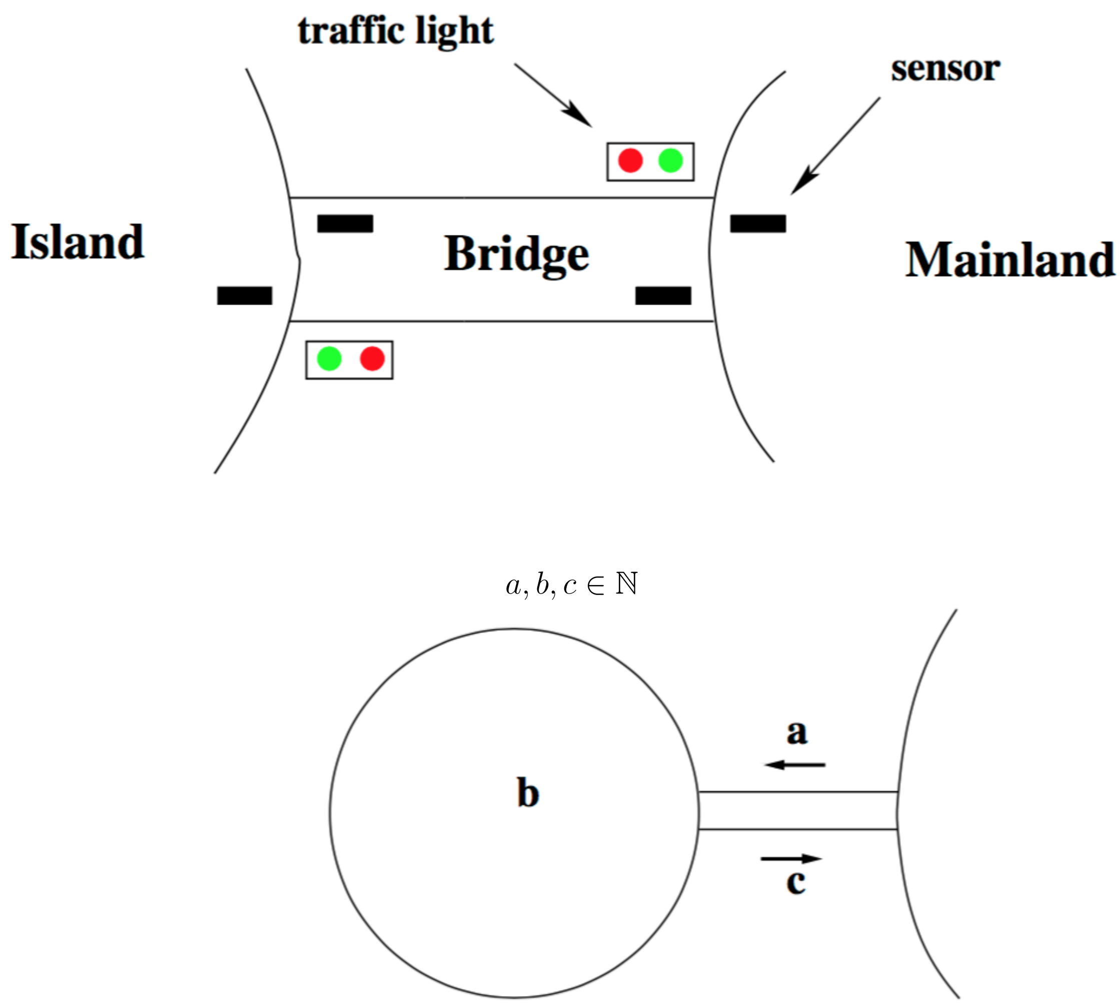
This example is motivated by the second refinement in Abrial's description of the bridge system: `http://deploy-eprints.ecs.soton.ac.uk/112/1/sld.ch2.car.pdf` (see slides 234-239 for a summary).

2 Informal Specification

The system controls cars on a bridge connecting the mainland to an island. The system is equipped with traffic lights to control the entrance to the bridge at both ends. It is assumed that cars only pass on to the bridge when the light is green. There are sensors and circuitry that provide measurements ($a, b, c \in \mathbb{N}$) of the traffic pattern in the system. There is a constant $d \in \mathbb{N}1$ that limits the total number of cars on the bridge and island at the same time, i.e. $a + b + c < d$. The bridge is one way, or the other, not both at the same time.

Question 1.

- What is the system boundary?
- How would you draw the context diagram?
- What are the monitored variables?
- What are the controlled variables?
- What types do we need?
- What are the E/R descriptions?
- What is the function table specifying the system?
- Are there any interesting Use Cases?



a denotes the number of cars on bridge going to island

b denotes the number of cars on island

c denotes the number of cars on bridge going to mainland

Figure 1: Controlling Cars on a Bridge

- Are there any important system invariants?

3 Difference between Specifications and Requirements

In the first part of Abrial's slides, he elicited the bridge requirements in a gradual fashion. We take his second refinement as a stopping point for this Lab.

In Abrial's slides, he provided events that model the *Plant* and the *Controller*, i.e. he is modelling the complete system (making certain assumptions about the operation of the Plant, i.e. the environment).

By contrast, in this Lab, we make no assumptions about signals coming from sensors in the Plant. For example, it is possible that drivers of cars might not respect the traffic signals, and they might enter the bridge illegally. We will thus make no assumption about the Plant.

Thus, we aim at providing a *Specification* of the bridge computer *Controller* in which we describe the relationship between controlled and monitored variables (with few restrictions on the monitored values). The Specification must thus deal with possible illegal and error inputs. In this sense, a *Specification* is a type of restricted form of *Requirement* at the shared interface between the Plant and Controller (i.e. the shared phenomena such as monitored and controlled variables).

A *requirement* states desired relationships in the Plant (environment) — relationships that will be brought about or maintained by the Controller. For example, that cars will not crash on the bridge. The requirement is concerned entirely with the environment, where the effects and benefits of the controller will be felt and assessed: the controller is purely a means to the end of achieving the required effect in the environment.

A *specification* describes the behaviour of the controller at its interface with the environment. Like a requirement, it is expressed entirely in terms of environment phenomena. Seen from the controller, a specification is a starting point for programming; seen from the environment, it is a restricted kind of requirement.

A specification is derived from a requirement. Given a requirement, we progress to a specification by purging the requirement of all features such as references to environment phenomena that are not accessible to the controller that would preclude implementation. The derivation is made possible by environment properties that can be relied on regardless of the controller's behaviour. These properties must, of course, be explicitly described if they are to be exploited.[1]

[1]The distinction between Specifications and Requirements is made by Michael Jackson, *Software*

4 PVS Specification and the Main Function Table

Question 2. Before we get started, can you write the function table(s) for the bridge controller?

Reflect before moving on.

Requirements & Specifications, 1995, ISBN 0-201-87712-0 (in Steacie Library). In the book page 20 the author tells the following story:

Some years ago I spent a week giving an in-house program design course at a manufacturing company in the mid-west of the United States. On the Friday afternoon it was all over. The DP Manager, who had arranged the course and was paying for it out of his budget, asked me into his office. 'What do you think?' he asked. He was asking me to tell him my impressions of his operation and his staff. 'Pretty good,' I said. 'You've got some good people there.'

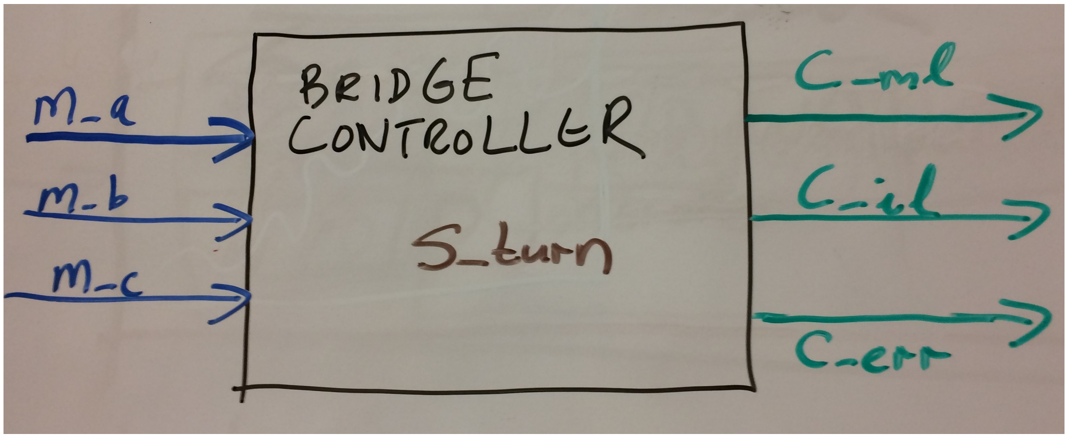
Program design courses are hard work; I was very tired; and staff evaluation consultancy is charged extra. Anyway, I knew he really wanted to tell me his own thoughts. 'What did you think of Fred?' he asked. 'We all think Fred is brilliant'.

'He's very clever,' I said. 'He's not very enthusiastic about methods, but he knows a lot about programming.' 'Yes,' said the DP Manager. He swivelled round in his chair to face a huge flowchart stuck to the wall: about five large sheets of line printer paper, maybe two hundred symbols, hundreds of connecting lines. 'Fred did that. It's the build-up of gross pay for our weekly payroll. No one else except Fred understands it.' His voice dropped to a reverent hush. 'Fred tells me that he's not sure he understands it himself.'

Terrific,' I mumbled respectfully. I got the picture clearly. Fred as Frankenstein, Fred the brilliant creator of the uncontrollable monster flowchart.

'But what about Jane?' I said. 'I thought Jane was very good. She picked up the program design ideas very fast.''Yes,' said the DP Manager. 'Jane came to us with a great reputation. We thought she was going to be as brilliant as Fred. But she hasn't really proved herself yet. We've given her a few problems that we thought were going to be really tough, but when she finished it turned out they weren't really difficult at all. Most of them turned out pretty simple. She hasn't really proved herself yet — if you see what I mean?'

I saw what he meant.



Abstract State Variable

The variable s_turn is an abstract state variable derived in Abrial's slides. It is an intermediate variable (neither monitored nor controlled) that helps us specify the computer controller. The specification (function table) is, after all, an **abstract** state machine. (This is not to commit the implementation to use s_turn)

The function table specification will be quite complex, so we will develop it in a hierarchical way with sub-tables. This will also help to make the check for completeness and disjointness **compositional**, i.e. the system function table is complete and disjoint if its sub-tables are complete and disjoint.

spec: [DTIME -> bool]

Monitored variables		**s_turn**	**c-ml**	**c_il**	**c_err**
i = 0		mainland	red	red	FALSE
i > 0	safe_input(i)	spec_pos(i)			FALSE
	¬safe_input(i)	NC	red	red	TRUE

-- Inputs are within safe bounds to preserve safety invariant
safe_input(i : POS_DTIME): bool ≜
∧ a(i) + b(i) + c(i) ≤ d
∧ a(i) or c(i) = 0
∧ ¬(c_il(i-1) = green ∧c_ml(i-1) = green)

To develop the specification of the bridge controller in PVS as a function table using T-ASM theory, we might start with the following:

```
bridge: THEORY
BEGIN
delta: posreal
IMPORTING Time[delta]

COLOR: TYPE = {green, red}
TURN: TYPE = {mainland, island}

d: upfrom(1) %nat1
% Monitored variables
m_a: [DTIME -> nat] % No of cars on bridge towards island
m_b: [DTIME -> nat] % No of cars on island
m_c: [DTIME -> nat] % No of cars on bridge towards mainland

s_turn: [DTIME -> TURN] % mainland and island pass variables

%Controlled variables
c_ml: [DTIME -> COLOR] %Mainland traffic light
c_il: [DTIME -> COLOR] %Island traffic light
c_err: [DTIME -> bool] %Error if something has gone wrong
```

Function Table Specification

```
% Precondition of bridge specification for i positive
% Inputs are within safe bounds
safe_input(i : POS_DTIME): bool =
   m_a(i) + m_b(i) + m_c(i) <= d
     % Maximum capacity obeyed
   AND (m_a(i)=0 OR m_c(i)=0)
     % Bridge is One-Way
   AND NOT (c_il(i-1) = green AND c_ml(i-1) = green)
     % traffic lights consistent in previous state

...

...

% Main Function Table
spec(i:DTIME): bool =
  COND
```

```
      i=0
    -> c_ml(i)=red AND c_il(i)=red
       AND s_turn(i)=mainland AND c_err(i)=False
     ,i>0 AND safe_input(i)  % inputs are within safe bounds
    -> spec_pos(i) AND c_err(i)=False
     , i>0
       %% NOT safe_input(i), i.e. inputs are not within safe bounds
      AND (   m_a(i) + m_b(i) + m_c(i) > d
           OR (m_a(i) > 0 AND m_c(i) > 0)
           OR (c_il(i-1) = green AND c_ml(i-1) = green))
    -> c_err(i) = True AND c_ml(i)=red AND c_il(i)=red
        AND  s_turn(i)=s_turn(i-1)
  ENDCOND
```

For simplicity, the main function table *spec* is written in terms of a sub-function table spec_pos, i.e. for time samples $i > 0$ (for safe inputs). The safe inputs are the invariants on the monitored variables elicited in the Abrial slides. The function table spec deals with the safe case as well as the error case.

What might spec_pos look like? From the Abrial slides something like this:

```
% Specification of Bridge Controller for i > 0
spec_pos(i:POS_DTIME): bool =
  safe_input(i)  % Needed for typechecking
  IMPLIES
  COND
    % when both lights are red
        c_ml(i-1)=red AND c_il(i-1)=red
     -> spec_red_red(i)
        % turn a light green
    % When a light is green
    % Turn mainland traffic light red
    ,     c_ml(i-1)=green
      AND c_il(i-1)=red
      AND s_turn(i-1)=island
      AND 0 < m_b(i) AND m_a(i)=0
     ->   c_ml(i)=red
      AND c_il(i)=red
      AND s_turn(i)=s_turn(i-1) %NC pass variables
    % Do nothing
    ,       c_ml(i-1)=green
        AND c_il(i-1)=red
        AND (s_turn(i-1)=mainland OR 0 = m_b(i) OR m_a(i) > 0)
```

```
   ->      c_ml(i)=c_ml(i-1)
       AND c_il(i)=c_il(i-1)
       AND s_turn(i)=s_turn(i-1)
% Turn island traffic light red
  ,      c_ml(i-1)=red
    AND c_il(i-1)=green
    AND s_turn(i-1)=mainland
    AND m_b(i)=0
   ->   c_ml(i)=red
    AND c_il(i)=red
    AND s_turn(i)=s_turn(i-1) %NC pass variables
  % Do nothing
  ,        c_ml(i-1)=red
       AND c_il(i-1)=green
       AND (s_turn(i-1)=island OR  m_b(i)> 0)
   ->      c_ml(i)=c_ml(i-1)
       AND c_il(i)=c_il(i-1)
       AND s_turn(i)=s_turn(i-1)
ENDCOND
```

Note that this function table is itself described in terms of another sub-function table `spec_red_red` (the case where both traffic lights are red). We leave `spec_red_red` for you to specify (by consulting the slides).

Also note that the sub-function table above has a precondition `safe_input`. This is because it will not type-check without this precondition (specified as an antecedent before the consequent that starts with "COND").

Question 3. Why do we need the precondition `safe_input`?

Question 4. Why do we need the "Do Nothing" cases?

4.1 Validation of the Specfication

You validate your specification by checking invariants and use cases.

- Check that spec and all its sub-function tables are well-typed, complete and disjoint.

- Check that important invariants hold
- Check important use cases

The following is the minimum you should check:

```
% Grind will not work
invariant: CONJECTURE
   (FORALl (i:DTIME) : spec(i))
     % Provided the spec is satisfied at all sampling instants
   IMPLIES
   (FORALL (i:DTIME) : NOT (c_ml(i) = green AND c_il(i) = green))
    %  lights will not both be green at the same time

j: DTIME
% mainland light turns green
% Grind might work
usecase1: CONJECTURE
       j > 0
   AND spec(j-1) AND spec(j) AND NOT c_err(j-1)
   AND c_ml(j-1)=red AND c_il(j-1)=red
   AND s_turn(j-1) = mainland
   AND m_a(j) + m_b(j) < d AND m_c(j)=0
   IMPLIES
        c_ml(j)=green AND c_il(j) = red
    AND s_turn(j) = island
    AND NOT c_err(j)

% Then, both lights turn red if no cars entering island
% and cars waiting to exit
% Grind might work
usecase2: CONJECTURE
       j > 0
   AND spec(j-1) AND spec(j) AND NOT c_err(j-1)
   AND c_ml(j-1)=green AND c_il(j-1) = red AND s_turn(j-1) = island
   AND 0 < m_b(j) AND m_a(j)=0
   AND safe_input(j)  %% inputs still within bounds
   IMPLIES
        c_ml(j)=red AND c_il(j) = red
    AND s_turn(j) = island AND NOT c_err(j)

% Then island light turns green
% Grind might work
```

```
usecase3: CONJECTURE
       j > 0
   AND spec(j-1) AND spec(j) AND NOT c_err(j-1)
   AND c_ml(j-1)=red AND c_il(j-1) = red AND s_turn(j-1) = island
   AND 0 < m_b(j) AND m_a(j)=0
   AND safe_input(j)  %% inputs still within bounds
   IMPLIES
        c_ml(j)=red AND c_il(j) = green
    AND s_turn(j) = mainland AND NOT c_err(j)
```

Question 5. What do the use cases check?

Yours might differ slightly but you should obtain something like this:

```
 Proof summary for theory bridge
    safe_input_TCC1......................proved - complete
    spec_red_red_pre_TCC1................proved - complete
    spec_red_red_TCC1....................proved - complete
    spec_red_red_TCC2....................proved - complete
    spec_red_red_TCC3....................proved - complete
    spec_pos_TCC1........................proved - complete
    spec_pos_TCC2........................proved - complete
    spec_TCC1............................proved - complete
    spec_TCC2............................proved - complete
    spec_TCC3............................proved - complete
    spec_TCC4............................proved - complete
    invariant............................proved - complete
    usecase1_TCC1........................proved - complete
    usecase1.............................proved - complete
    usecase2.............................proved - complete
    usecase3.............................proved - complete
    Theory totals: 16 formulas, 16 attempted, 16 succeeded (0.00 s)

Grand Totals: 16 proofs, 16 attempted, 16 succeeded (0.00 s)
```

5 Writing a Requirements Document

What might a Requirements document might look like.

- Where is the System Boundary?
- What are the monitored variables? What are their types?
- What are the controlled variables? What are their types?
- Specify a complete and disjoint function table that describes the input/output behaviour of the SUD? (You might want to draw this table in your requirements document to help with the PVS specification.)
- Now (a) write out the atomic R-descriptions for the plant (number them) (b) state what the monitored variables are and in a different table what the controlled variables are and (c) draw the function table for the car interlock system and provide evidence that you have validated the function table.
- We will be using Latex to prepare documentation for the assignment and project. You may try to prepare the document using Latex. See `https://wiki.eecs.yorku.ca/project/sel-students/p:tutorials:latex:` (login). There is a link to a Latex table generator.
- You are not required to use Latex for this Lab. Use any documentation preparation system you like provided it is neat.

Ensure that it is neat! Ensure that it is minimal! It does not require more than half a page in a large font. Too many cooks spoil the broth (i.e. too many rows and not enough organization and thought spoil the function table).

6 What will be in your document

Understand and agree upon what users want before attempting to create solutions.

Finding out what is needed instead of rushing into presumed solutions is the key to every aspect of system development. Most technical problems can be solved, given determination, patience, a skilled team—and a well-defined problem to solve.

A precise requirements document will contain all the information needed by the developers to build the system wanted by the users—and no more (i.e. it should not be polluted with design and implementation detail).

It is important to write the informal requirements atomically (with a number to track the requirements), e.g.

REQ1	The bridge is one way or the other but not both at the same time

The above requirement will be checked using the invariant conjecture in PVS.
Your document will contain:

- Informal statement of the problem
- Context diagram
- Table of monitored variables with its types and another table of controlled variables with their types.
- Atomic requirements.
- The function table that is complete and disjoint.
- Various use cases (one is enough for this introductory document)
- The PVS specification of the function table and validation of completeness/disjointness, invariants and use cases. We showed one use case, but in general there will be many.