Informatics 1 - Computation & Logic: Tutorial 1

Computation: Transducers

Week 3: 1–5 October 2018

A *finite state machine (FSM)*, is an abstract model of computation that can be used to design hardware and software. For example, FSMs can represent the behaviour of devices such as vending machines, elevators or traffic lights, or software systems such as an online store.

There are many flavours of FSM. We will start with deterministic transducers. These machine have states, inputs and outputs. They are used to model many embedded systems—and in many other areas. Here we start by modelling some simple vending machines. The user can insert coins and press buttons. The machine responds with a sequence of outputs–giving drinks, and giving change.

In this tutorial you will design some transducers, and implement their behaviours in Haskell. You should begin by reading the introduction.

The tutorial activities for this week include work on the Cruise Control controller. You should read the appendix describing this system in advance of the tutorial.

You have four tasks, first to read the Introduction and appendix, then to complete three sections of exercises. As a very rough guide, I expect you to spend roughly 45 minutes on each of these four tasks. If this estimate is wildly wrong, please let me know.

Michael Fourman

0 Introduction

Suppose we want to program a machine, for example, a vending machine. Before we get down to programming, it would be useful to come up with some general plan of how it should operate. This should be based on a specification - a description of features and behaviours the machine is meant to have.

Vending machines can be simple, like this: https://www.youtube.com/watch?v=WnZ6PYK2yyU

Or more complicated, as in this example:

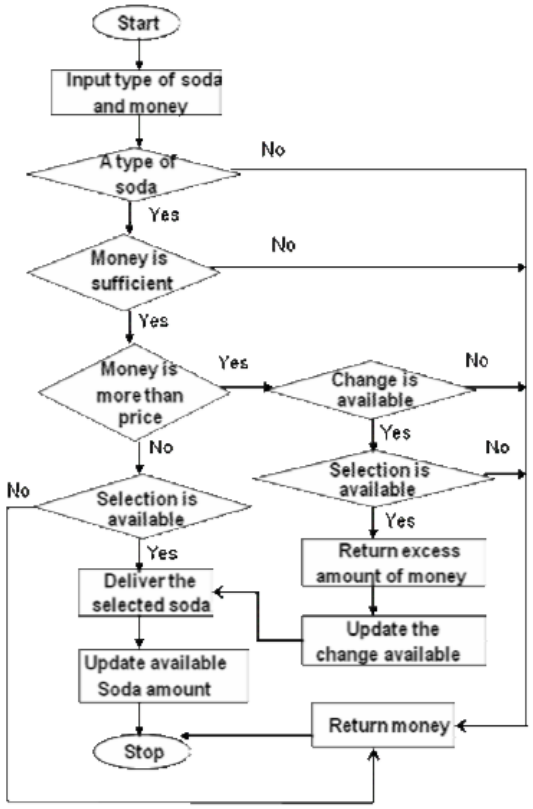


Fig. 1. Flow chart of the operation of a Soda Vending Machine.

The flow chart in Fig. 1 is copied from a paper *Towards software test data generation using binary particle swarm optimisation*, Agarwal et al. **XXXII NATIONAL SYSTEMS CONFERENCE, NSC 2008**, describes the operation "buy Soda" in a Soda Vending Machine. The flow chart includes one primary and two alternate scenarios. In the primary scenario, the customer makes a cash (money) input (taken to be an integer amount here), and then makes a soda selection (an integer value). The soda machine then dispenses a cold can of the selected soda. The alternate scenarios are the "out of soda selection" scenario and the "out of change" scenario. In an occurrence of any of the alternate scenarios, the current transaction is terminated and the money is returned. "Out of soda" scenario occurs when the soda type, the customer demands, is not in the menu or not present in the inventory. The other scenario "out of change scenario" occurs when the required change is not available in the cash reserved for returning the change.

We will take a different approach. Rather than trying to specify the decisions the machine must make, we model the way it changes state and generates outputs in response to inputs.

We start with a very simple example then implement the machine in Haskell. In the beginning, our very simple machine contains one Caramel Wafer. It has one button. When the button is pressed, it outputs the Wafer; then it is empty.

The machine has two states: `Full` and `Empty`.

We can represent it with a diagram.

```
data Input  = Button
data Output = Wafer
data State  = Full | Empty

machine :: State -> Input -> ([Output], State)
machine Full Button = ([Wafer], Empty)
```

This machine is a transducer, with states, inputs and outputs. A transducer steps from one state to the next state in response to an input, and may produce an output. The states are represented as nodes in a directed graph. The possible transitions from once state to another are represented by the edges of the graph. These are drawn as arrows from one state to another, labelled with `input > output`.

This behaviour is summarised by the Haskell function

```
machine :: State -> Input -> ([Output], State)
```

Given the current state and an input, this function returns the output and next state.[1] Thus the machine is completely described by the sets of states, inputs, and outputs, the *transition function*, `machine`, and the start state.

We can see what happens when the machine is in the start state `Full` and we press the `button`. The machine delivers a Wafer and moves to its `Empty` state.

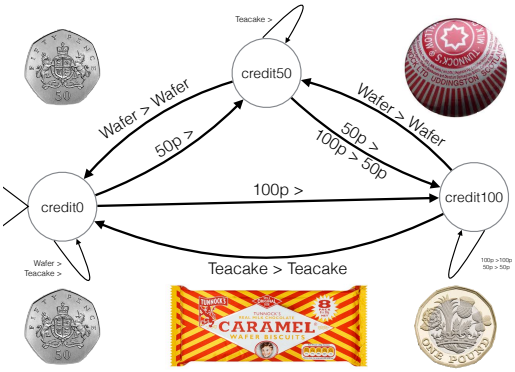
```
Prelude> :load machine
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> machine Full Button
([Wafer], Empty)
```

* * * * * * *

Our next machine has three states, corresponding to the amount of the user's credit.

This machine only accepts 50p or £1, coins and has two buttons, `Teacake` and `Wafer`.

The machine has an *initial* or *start* state – with zero credit. A credit of 50p makes `Wafer` available; a credit of £1 makes `Teacake` or `Wafer` available. The machine gives change.



We will use Haskell to model the behaviour of this machine. First we introduce enumerated types for the states, inputs, and outputs. In this example, we use the same enumerated type `InOut` for both inputs and outputs; `Input`,and `Output` are *synonyms* for this type.[2]

```
data InOut  = Fifty | Pound | Teacake | Wafer        deriving(Show)
type Output = InOut -- change or product dispensed
type Input  = InOut -- money inserted or button pressed
data State  = Credit0 | Credit50 | Credit100         deriving(Show)
```

[1] We use a list, `[Output]`, rather than just an `Output` value, since for some transitions there may be no output. You will meet a cleaner way of doing this in Haskell later in the course.

[2] Enumerated types are the simplest user-defined types in Haskell. We simply use the keyword `data`, introduce a new type name, and list the values we want to introduce. We can use the values in function definitions as in these examples The incantation, `deriving(Show)`, makes ghci able to show such values.

The machine is then modelled by a function

```
machine :: State -> Input -> ([Output], State)
```

that, for each current state, the outputs and next state corresponding to each input,: [3]

```
machine Credit0   Fifty   = ([]       , Credit50)
machine Credit0   Pound   = ([]       , Credit100)
machine Credit0   _       = ([]       , Credit0)
machine Credit50  Fifty   = ([]       , Credit100)
machine Credit50  Pound   = ([Fifty]  , Credit100)
machine Credit50  Wafer   = ([Wafer]  , Credit0)
machine Credit50  _       = ([]       , Credit50)
machine Credit100 Teacake = ([Teacake], Credit0)
machine Credit100 Wafer   = ([Wafer]  , Credit50)
machine Credit100 coin    = ([coin]   , Credit100)
```

Now we are interested in how the machine will respond to a *sequence* of inputs. We can build Haskell functions to examine its behaviour. For example, we can write a function

```
finalm :: State -> [Input] -> State
```

such that, if we start the machine in some state `s` and give it a sequence, xx, of inputs,[4] `finalm s xx` computes and returns the final state of the machine.

```
finalm s [] = s -- with no input we stay put
finalm s (input : moreInputs) = finalm s' moreInputs -- s' is next state
  where (_, s') = machine s input  -- we don't care about the outputs
```

A simple transformation generalises this to a function that takes the machine as a parameter. We use type variables to create a general function applicable to other examples.

```
final :: (state -> input -> ([output], state)) -- note the type variables
      -> state -> [input] -> state             -- for state,input, output
final m s [] = s -- with no input we stay here
final m s (input : moreInputs) = final m s' moreInputs
  where (_, s') = m s input
```

We will use two more functions that take a machine's transition function as a parameter.[5]

```
trace :: (state -> input -> ([output], state))
         -> state -> [input] -> [state]
```

traces the sequence of states visited for a given sequence of inputs.

[3] The underscore pattern, `_`, matches any value. In the last line of this code, `coin`, is a variable. Since `Teacake` and `Wafer` are dealt with already, the only values this pattern will match are the coins `Fifty` and `Pound`; the output `[coin]` will return whichever coin was inserted.

[4] We adopt the convention that the machine takes successive inputs from the head of this list, then continues to process the tail. So the first input is at the head of the initial list.

[5] These functions return (respectively) a list of states visited, and a list of outputs produced. For these lists, we adopt the same convention, that first state visited, and the first output produced come at the head of the result lists. This means that we can pass the outputs of one machine to be the inputs of another.

```
out :: (state -> input -> ([output], state))
       -> state -> [input] -> [[output]]
```

records the sequence of outputs produced for a given sequence of inputs, starting from a given state. Here is the code: (equivalent to `final machine`).

```
final :: (state -> input -> ([output], state)) -- note the type variables
      -> state -> [input] -> state             -- for state,input, output
final m s [] = s -- with no input we stay here
final m s (input : moreInputs) = final m s' moreInputs
  where (_, s') = m s input

trace :: (state -> input -> ([output], state))
         -> state -> [input] -> [state]
trace m s [] = [s] -- with no input we stop here
trace m s (input : moreInputs) = s : trace m s' moreInputs
  where (_, s') = m s input

out :: (state -> input -> ([output], state))
       -> state -> [input] -> [[output]]
out m s [] = []
out m s (input : moreInputs) = o : out m s' moreInputs
  where (o, s') = m s input  -- here we need the outputs
```

We can apply these functions to any machine defined, in the right way, in Haskell.

This kind of FSM, with inputs and outputs, and exactly one transition from each state for each label, is called a *deterministic transducer*.

Finite state transducers are useful for modelling systems like the vending machine. They also have major applications in natural-language processing ranging from morphological analysis to finite-state parsing. For example, the analysis and generation of inflected word forms can be performed efficiently by means of lexical transducers.

1 Vending machines

1. This question concerns the three-state machine introduced above.

   (a) Try to answer this question *without* running the Haskell code. Then check your answers by running the code. Assume that the machine starts in the start state, and is given the following sequence of inputs:

   ```
   [Fifty, Pound, Wafer, Fifty, Teacake, Teacake]
   ```

   - What is the final state reached?
   - What is the trace of this computation?
   - What sequence of outputs is produced?

   (b) How can we check that for *any* sequence of inputs the total value of the outputs will always be less-than-or-equal to the total value of the inputs?

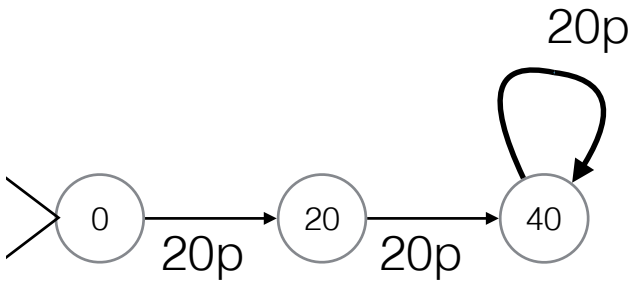
2. This finite state machine could be used as part of a vending machine. It accepts any sequence of 20p coins and models a credit of up to 40p.

   (a) How does this machine deal with input of more than 40p?

   (b) Modify the machine by adding outputs to return change and to also allow 10p coins.

   (c) Implement the transition function for your transducer in Haskell.

   (d) Use the functions trace and out to test the behaviour of your machine on various sequences of inputs.



3. This question concerns the design of a a hot drinks machine. The machine should accept 10p, 20p, and 50p coins, and sell tea for 50p and coffee for 70p.

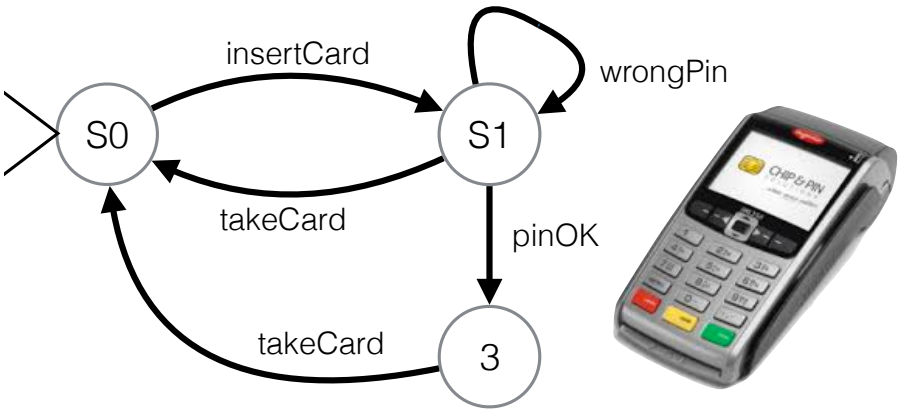
   (a) Design a transducer that could be used to control this machine. After a successful sale the transducer should return to its start state.

   (b) Implement the transition function for your design in Haskell. How can you test your design?

   (c) Consider your answer to part (a). Can you modify your machine so that it keeps track of its coins, and only gives change when it has the right coins available?

2 Card payments and ATM

4. The diagram below is a first step in modelling the control logic of a chip & PIN card payment terminal. It represents a single transaction, ending in an accepting state if the transaction is successful. Note that the transaction will fail if the card is removed too early.



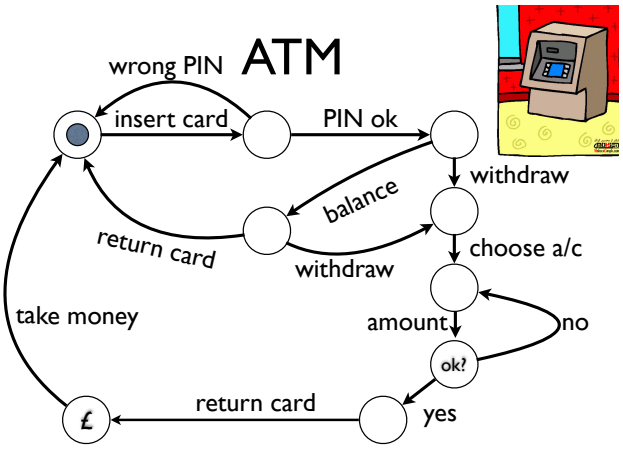
The terminal interacts with the user and communicates with the bank to check the PIN. When the user enters a PIN it is stored internally the control logic can issue a `pinok` query to send this to the bank, The bank responds with `ok` or `nok`.

(a) Refine this design by specifying inputs and outputs, and adding new transitions if necessary, to produce a transducer that models this controller.

(b) Modify your machine so that the transaction will also fail if the wrong PIN is entered three times.

(c) Write a Haskell implementation of the transition function for your machine, and test your design.

5. Consider the initial design for an ATM controller sketched here.



   (a) Refine this design. Introduce a suitable set of inputs and outputs and produce a transducer that models the ATM controller.

   (b) Modify the PIN-checking section of the ATM controller to allow more than one attempt to enter the PIN. Your machine should retain the card after three wrong PIN entries.

   (c) Implement your transducer in Haskell and produce a set of tests to check its working.

3 A big machine

6. Consider the cash-input section of a checkout machine that accepts coins of 1p, 2p, 5p, 10p, 20p, 50, £1, £2. It can store up to 100 of each of these coin types, which keeps track of a user credit of up to £50. So its state consists a six-tuple of numbers less than 101 – recording the number of coins or each type, and two further numbers, from zero up to five thousand, recording the amount due and user credit in pence.

   This is a finite state system, but it's too big for a useful diagram. Write the transition function for this system in Haskell. What would be used as the start state for your system?

   An input to the cash-input section should specify the total due (any number of pence up to £50). The cash-input section should then collect payment from the user; give change; and output either `OK` or `NOK`.

   `OK` signifies a successful transaction – the user has paid successfully. `NOK` signifies that for some reason the transaction is incomplete – in this case the user's money should be returned.

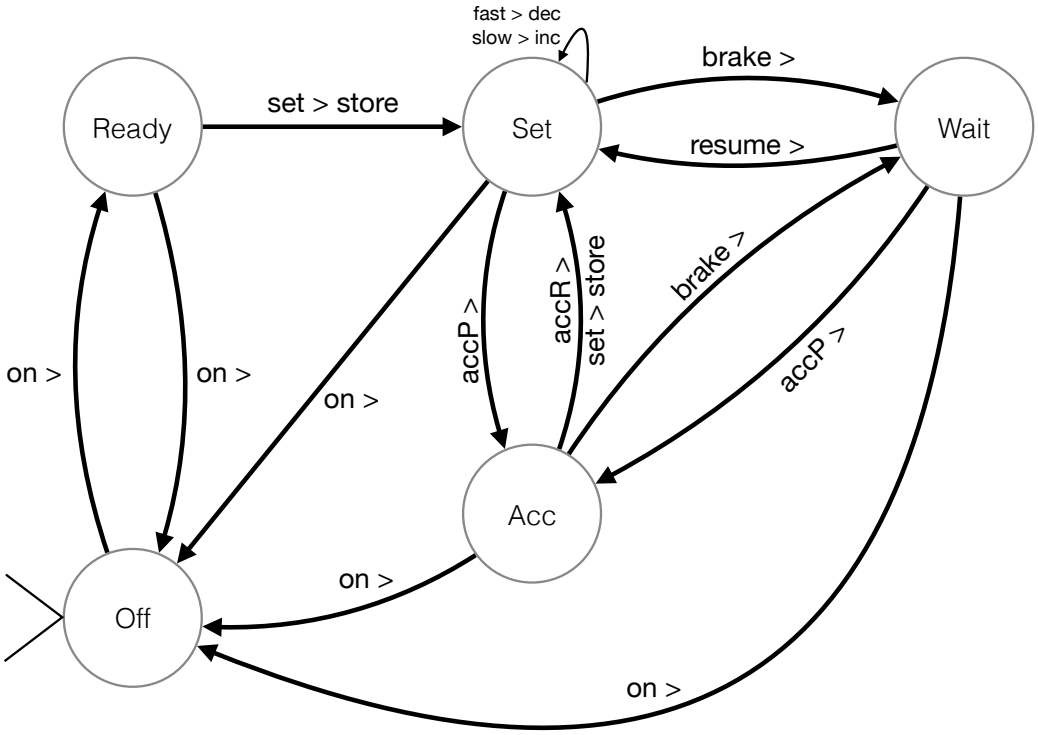
   You may replace `NOK` with a collection of different error codes to make the machine more user friendly.

Tutorial Activity

As usual, you should start by working as a group to identify and resolve any problems with the homework. We will spend the first half of the tutorial working through the homework.

Appendix: Cruise Control

Consider the Cruise Control introduced in class. The system is described more full as an appendix to this note. We describe the transducer diagrammatically, the labels on the arcs represent inputs. If the label has the form *in* > *out* then output *out* is generated on the transition for input *in*. In this diagram, where we omit transitions from a particular state for some input we intend it should be a loop back to the same state with no output.



1. First code the transition function for this system in Haskell. You can do this as a group effort.
2. Design a set of test sequences to check that the system meets the specification (given below). Can you demonstrate that each of the four requirements is satisfied?
3. If you find that the design given fails to meet one of the constraints, alter the design to produce one that satisfies all the constraint.
4. Consider the scenario reported under Evolution/Maintenance below. Can you produce a sequence of inputs that mirrors this scenario?

5. What further constraint should be added to the specification to avoid this kind of accident?

6. Implement and test a modified design that satisfies your new constraint (in addition to those already listed).

Cruise Control

On long car journeys drivers find it very tiring to keep up continuous pressure on the accelerator pedal. To avoid the need to do this many cars now have a system called *cruise control*. This is a controller that sits between the driver and the controls of the car (here the throttle that determines how fast the car travels). The cruise control system allows the driver to set a particular speed and then the controller maintains that speed until the driver changes the speed, uses the brake, or switches the system off.

These systems are usually controlled by a number of push-buttons on the dashboard or steering wheel of the car. The system usually has different modes of operation depending on the history of how the car has been controlled. The modes of operation (and subsidiary states arising in a particular mode) are usually modelled by a finite state machine and the meaning of the actions on the push buttons are given by saying what transitions take place between the states of the FSM when the buttons are pressed.

Specification

It is common practice in the world of embedded controller to use a very low level specification which is close to the implementation of the system. This practice arises out of the evolutionary approach to the design of such controllers. The next controller is very often just like the previous one with more bells and whistles.

Ideally the specification for a system should describe what the system should do – not how it does it. Here we list the main properties we would expect of the system. A real world system would have one or two extra features but would be quite close to this.

1. The driver should always be able to turn the system off.

2. The driver should be able to request the system to maintain the current speed.

3. The system should not operate after braking

4. The system should allow the driver to travel faster than the set speed by using the accelerator.

Design

Taking account of current practice in the industry (and hence user expectations of how the system will work) we can give a preliminary design that we believe matches the specification. The design consists of three components: the inputs and outputs to the system, the modes

(or states) of the system, the effects of the inputs on the system in each state. Here we should expect to say explicitly what the effect of an input is for every state.

The inputs are:

- `on`: on/off button
- `set`: set the cruise speed to the current speed
- `brake`: the brake has been pressed
- `accP`: the accelerator has been pressed
- `accR`: the accelerator has been released
- `resume`: resume travelling at the set speed
- `correct`: indicates the car is travelling at the correct speed.
- `slow`: indicates the car is going slower than the set speed
- `fast`: indicates the car is going faster than the set speed

The outputs are:

- `store`: store the current speed as the cruise speed
- `inc`: increase the throttle
- `dec`: decrease the throttle

The main states (or modes) of the controller are:

Off: The system is not operational.

Ready: The system is switched on but so far no speed has been set to cruise at.

Set: A cruise speed has been set and the system is maintaining it.

Wait: The system has a set cruise speed but at some time the driver used the brake and caused the system to wait until the resume button is pressed to bring the car back into cruise control.

Acc: The system has a set cruise speed but the accelerator has been pressed to override cruise control until the accelerator is released.

Testing

In testing the system we might first want to explore the behaviour of the system to see if we can discover anomalous behaviour. To do this we might imagine using the FSM to derive sequences of actions. In testing we usually use a *coverage criterion* to limit how much testing we do. Because the system is very small we might want to use an "all paths" criterion where the set of test inputs covers all the possible paths through the finite state machine (for very large machines we can't do this).

We would also want to check that all the properties in the specification are satisfied by the design. Here we can see that they are satisfied by the design. For more assurance we might want to express the properties in the specification in some formal logic that would allow mathematical verification of the properties in principle.

We might also consider the transitions from each state in turn and check that their effect is correct. In our example consider the effect of a `set` operation in the *Wait* state. Here we might require that we also do a `store` output to reset the cruise speed. We might also argue that instead of a loop the arc labelled `set`/`store` might go to the *Set* state rather than looping on *Wait*.

Evolution/Maintenance

After some months of operation the following accident report arrives at the car company[6]:

> *The incident occurred when the driver was on a highway on a rainy night. The traffic was slow, travelling at about 40 mph. The driver engaged cruise control and set it to 40 mph. Later the rain cleared and the traffic got faster so the driver used the accelerator to increase the speed to 60 mph and travelled in this mode for some miles (the controller still in set mode but overridden by the accelerator).*
>
> *Coming to the exit ramp the driver turned off and released the accelerator to coast up the ramp. At that point the cruise control aimed to stabilise the speed at the set level (40 mph). The driver was taken by surprise, had to brake suddenly and lost control of the car which travelled through a stop sign. Fortunately no accident occurred.*

During the life of a product many accident reports are lodged with the manufacturer. These are usually analysed to see if there is some systematic source of accidents that can be eliminated if it presents a significant threat. The problem identified here is just such a case. One possible solution is to treat the accelerator like the brake, and require the driver to explicitly press the resume button in order to return to cruise control after accelerating. This would require the controller software to be amended and updated in vehicles, probably by replacing the controller chip.

This tutorial exercise sheet was written by Dagmara Niklasiewicz and Michael Fourman, drawing on material from an earlier tutorials produced by Stuart Anderson and Murray Cole. Send comments to `Michael.Fourman@ed.ac.uk`

[6]This example is courtesy of *Do you know what mode you're in? An analysis of mode error in everyday things*, Anthony Andre (Interface Analysis Associates, San Jose, CA) and Asaf Degani (San Jose State University, CA)