Investigating post-completion errors with the Alloy Analyzer

Technical Report CS-RR-433

Russell Boyatt and Jane Sinclair
University of Warwick
United Kingdom

Abstract

Post-completion errors are a particular kind of error found in interactive systems. This type of error occurs through the incorrect sequencing of goals and sub-goals, when the primary goal is achieved before all of the prequisite sub-goals have been satisfied. This paper shows how we can check for this property in a formal model of an interactive system. Specifically, we suggest that lightweight formal methods, such as the Alloy structural modelling language, are particulary well suited for this task. As a case study we develop two example interactive systems. The first is the ubiquitous chocolate machine, where both the chocolate and change must be delivered to the customer. The second model is of a typical cash machine and explores the problems of returning the cash and the cash card in the correct order. Both of these models are developed in the Alloy language.

Also available from `http://www.dcs.warwick.ac.uk`.

Contents

1 Introduction

1.1 Post-completion errors

A common error with interactive system is a specific type of behaviour familiar to those who have left their cash card in an ATM after withdrawing cash. Retrieving the duplicate copies from a photocopier but forgetting to collect the original documents is another example of the same problem. These types of errors are known as a "post-completion" errors. The key characteristic of these errors is that the primary goal is realised before all of the sub-goals have been satisfied. In order words, we achieve our main objective but, perhaps due to the sequencing of events, forget to do the items that were before it on the list. Post-completion errors do not always occur, in that we cannot assume that if they *can* occur they will do – most of the time you remember the original documents on the photocopier plate. However we aim to prevent even the possiblity of these errors occuring. In general, post-completion errors can be avoided by looking at sequences of sub-tasks and ensuring they must all occur before the main task at the end of the sequence. Although it can be trivial to identify some instances of this error we can make no guarantees without a rigorous formalised approach. If we can correctly formulate the problem we can use existing formal methods tools to check for the various interactive errors. The approach we describe in this paper focuses on the use of a lightweight formal method, namely Alloy. Although we could perform the analysis with a traditional formal method, we are interested in the lower barrier to entry afforded by lightweight formal methods. Jackson and Wing [JW96] emphasise the differences between lightweight methods and a more traditional approach. They characterise the respects in which a lightweight method will be limited in its scope as: partiality in language; partiality in modelling, partiality in analysis and partiality in composition. The trade-off here is that a formal basis can be used to provide automatic feedback in a more focused way in order to direct an ongoing development. Tools can be used by developers who are not necessarily formal methods experts and may provide a more "programmer-friendly" interface. Such approaches are likely to be used much earlier in the development process to explore requirements and to validate aspects of an evolving specification. We focus here on post-completion errors as an example property but is chosen only to serve as an example and our models are not designed solely for any particular property. The models are general and able to be used to investigate many other properties (e.g. user interface forgiveness).

1.2 Alloy language and the Alloy Anaylzer

Alloy [Jac06] is a structural modelling language inspired by the state-based Z notation [Spi89]. It allows the user to express complex structures and constraints using a relational logic, combining first-order logic with relational calculus. However, the syntax of the language is specifically designed to be accessible and understandable, thus facilitating the development of formal models in a notation more recognisable to non-specialists.

Basic structures in the model are described using *signatures*. Relations within these signatures are called *fields*, which describe the relationship between the signatures. Logical expressions constrain further the possible instances of

a model. These predicates are also used to define operations by describing the state change associated with the operation. This allows the specifier to capture and operational view of the model's dynamic behaviour and traces can be defined and explored in the system.

One of the main criticisms of formal methods is the overemphasis on full formalisation – that it is unnecessarily expensive and restrictive. The approach of lightweight formal methods attempts to counter these criticisms by allowing for a focused and partial specification of a system. By aiming to reduce the complexity of the specification process we allow for a greater flexibility and degree of experimentation that would be difficult and expensive to achieve with a full formalisation process. Alloy is one example of the lightweight approach to formal methods. It retains many of the benefits of formalisation and also offers a greater agility and capacity for exploration not otherwise possible.

The Alloy Analyzer[1] is a tool developed by MIT to support the Alloy language. Alloy uses the SAT-based model finder Kodkod and offers a fully-automatic means to analyse models that is in contrast to theorem proving or model checking approaches. The model-finding is performed within user-specified restricted scopes, limiting the size of instances of each model. This does limit the results as it would not be able to identify counterexamples greater than the biggest scope we analyse. Jackson ([Jac06], p. 141) justifies this in terms of the *small scope hypothesis*, that is far more effective than traditional testing and that "most bugs have small counterexamples". Unlike traditional testing, Alloy supports exhaustive checking for such counterexamples within a small scope.

Using the tool, models written in the Alloy notation can be interactively interrogated and user-specified properties can be checked. The capacity for incremental analysis and development allows the feedback received to be interpreted and then incorporated into the model. An example of the output provided by the Alloy Analyzer when in operation is shown in Figure 1.

2 Example: Chocolate Machine

Our first example is a very simple chocolate machine. In the first version of our chocolate machine we follow an example similar to the one given by Curzon and Blandford [Bla00]. The chocolate machine accepts two pence coins, and is capable of returning one pence coins. It dispenses only one type of chocolate, which costs one pence each. When a coin is inserted, the customer must push a button to receive the chocolate and must then push another button to receive their change. We make some simplifying assumptions: the machine always has sufficient chocolate and change, and that when a user pushes the button for a chocolate they actually collect the chocolate from the dispenser (and similarly for change). We could consider the errors that occur when these tasks are interrupted and problems occur but these tasks are not the post-completion errors we are to consider.

Through our interaction with the chocolate machine, our primary goal to retrieve the chocolate and our secondary goal, or sub-goal, is to retrieve the change. In this situation we would regard a post-completion error as occuring when the user retrieves their purchased chocolate but neglects to collect their

[1] Available from `http://alloy.mit.edu`

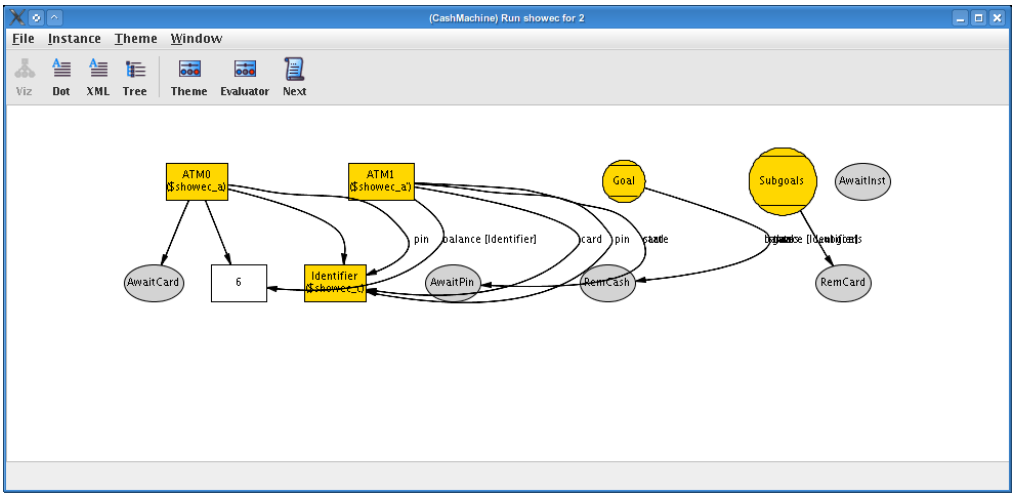

Figure 1: Example output from the Alloy Analyzer

change money. Our first Alloy-based model of the chocolate machine is as described above and is shown in appendix A. This model has an obvious post-completion error as the change is always returned after the chocolate - allowing the user to retrieve the chocolate before retrieving the change. We can discover this error both through our own examination of the situation and also by using the Alloy Analyzer to help us examine a model of the scenario.

2.1 Dissecting the chocolate machine model

We construct a model that is representative of the chocolate machine described. Initially, we must decide the possible states of the chocolate machine. The machine can be either: waiting for money to be inserted (we label this the 'Reset' state); having accepted money and waiting for the chocolate button to be pushed (the 'Coin' state); after a chocolate has been dispensed (the 'Chocolate' state); or, having returned change to the customer (the 'Change' state). We use a signature declaration as follows:

```
abstract sig ChocState {}
one sig Reset, Coin, Chocolate, Change extends ChocState {}
```

The state of the chocolate machine consists of its current state as defined above, the amount of money that has been entered by the customer and the type of chocolate, if any, in the dispenser. We declare a signature, Choc, to hold the state of the chocolate machine at one instant in time.

```
sig Choc {
  balance: one Int,
  state: one ChocState,
  op: OP,
  dispenser: lone ChocType
}
```

There is one additional field in the `Choc` signature we have not yet mentioned - the field labelled 'op'. So far, we have considered the model from the perspective of the chocolate machine and not with regard to the customer and their interaction with the machine. If we consider the interactions the customer will have with the machine, we can identify several actions the customer may perform: entering a coin, choosing a chocolate and requesting change. To help illustrate the evolving scenario we declare an additional signature to record what the *customer* is doing to initiate changes in the state of the chocolate machine. At each stage in the trace, we will be able to observe the actions of the customer. This information is purely for illustrative purposes and aids our understanding of examples produced by the analyser.

```
abstract sig OP {}
one sig ENTERCOIN, PUSHCHOC, PUSHCHANGE, RESET extends OP {}
```

To represent the evolving state of the chocolate machine we must constrain the possible states of our model. Predicates are used to represent operations on the state of the chocolate machine and describe the relationship between two instances of the `Choc` signature. We can think of these as the 'before' (`c`) and 'after' (`c'`) states. We present the predicate that presents the customer, who has just inserted a coin, selecting the chocolate to purchase:

```
pred buychoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(1)] &&
  c'.balance = INT/sub[c.balance, int(1)] &&
  no c.dispenser &&
  c'.dispenser = Choc1
  c'.state = Chocolate
}
```

This predicate, although it represents an operation, is a constraint. It constrains the state before selecting the chocolate (`c`) and constrains the state after selecting the chocolate (`c'`). For example, "`c.state = Coin`" ensures that the chocolate machine is in the 'Coin' state initially and then "`c'.state = Chocolate`" ensures that it is in the 'Chocolate' state afterwards. This is a familiar and expressive approach in state-based specifications facilitating the use of Alloy. As we are considering the amount of money that has been entered into the machine, we use the integer operations in the `util/integer` package using them with the prefix 'INT'. We must have sufficient funds available to purchase the chocolate and we must subtract the cost of the chocolate from the available balance. We also represent the chocolate being dropped into the (empty) dispenser. Similarly, other operations on the state of the chocolate machine, such as entering coins and returning change, are described using this approach to predicates.

We aim to examine the dynamic behaviour of the chocolate machine and must therefore introduce a mechanism to capture the evolving state of the chocolate machine but still allow for the model to be statically checked. Alloy already has this mechanism in the form of traces. The predicate describing the valid traces in the system is:

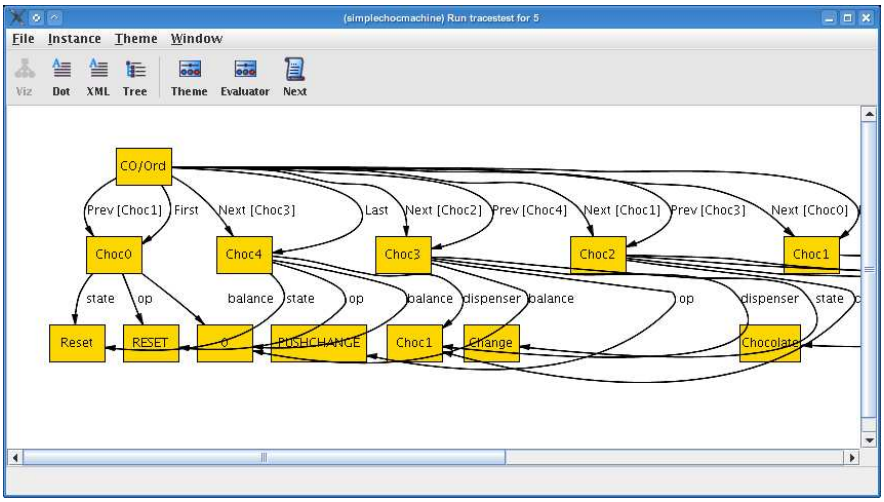

Figure 2: Example traces output for the chocolate machine.

```
pred traces {
        init[CO/first[]] &&
        all c:Choc-CO/last[] | let c' = CO/next[c] |
                    ((entercoin[c,c']  && c'.op = ENTERCOIN)
                     or
                     (buychoc[c,c']  && c'.op = PUSHCHOC)
                     or
                     (askchange[c,c']  && c'.op = PUSHCHANGE)
                     or
                     (reset[c,c']  && c'.op = RESET))
}
```

This says that the initial condition holds for the initial time step, and then for all subsequent times the machine must change in accordance with one of the four predicates: `entercoin`, `buychoc`, `askchange`, and `reset`. Additionally, the actions of the customer are defined here to help annotate any examples returned. The four actions by the customer are: entering a coin (ENTERCOIN), pushing the chocolate button (PUSHCHOC), pushing the change button (PUSHCHANGE) and removing their change from the machine (RESET). An example of the traces output from the Alloy Analyzer is shown in figure 2. The ordering library module we use here is generic, usable with any signature type and not specific to this example.

2.2 Checking for post-completion errors

To be able to check for post-completion errors we must define the sequences of events that represent a single interaction of the customer with the chocolate machine. For the purposes of our model, we will refer to a single interaction as a 'transaction', consisting of a trace that begins with the initial state (awaiting coins) and finishes with the machine resetting for the one and only time in that

transaction.

```
pred transaction {
        traces &&
        (CO/last[]).op = RESET &&
       RESET !in (Choc - CO/last[]).op
}

one sig Goal {goals : set ChocState} {goals = Chocolate}
one sig Subgoals {subgoals : set ChocState}{subgoals = Change}

assert goalsmet {
  transaction => let m = CO/max[state.(Goal.goals)] |
    some m => all sg: Subgoals.subgoals | state.sg in CO/prevs[m]
}
check goalsmet for 5
```

Also shown above is the mechanism for representing the goal and sub-goals, the order in which they are satisfied being of great importance to the post-completion errors we are considering. We define the main goal to be the dispensing of the chocolate, and the sub-goal to be the dispensing of the change. For a post-completion error to occur in our model, the chocolate would be dispensed *before* the change is dispensed. In the `goalsmet` assertion, we state that for valid transactions all subgoals (`sg`) must occur in states before the state in which the goal occurs (`m`). Assertions are something we believe to be true about our model and the `check` command asks Alloy to provide us with the counterexamples, demonstrating that it is possible for the primary goal to be achieved prior to all of the sub-goals being satisfied.

Now we have constructed the model and introduced the appropriate mechanisms to represent and check for post-completion errors, we can begin to explore the properties of the model using the tool itself. Checking the `goalsmet` assertion quickly provides a counterexample that should, in this case, already be easily identifiable and simple to confirm by a visual inspection of the model. The current version of the model is constructed in a way that prevents the machine from engaging in behaviour where the change is dispensed before the chocolate. Addressing this post-completion error suggests modifications to the actual chocolate machine. Of course, any chocolate machine that requires a button requesting change after the chocolate dispensed will have a post-completion error. Perhaps a solution is to, after the chocolate has been selected, noisly return the change to the customer just before the chocolate itself is dispensed.

2.3 Extending the chocolate machine

We will now consider several possible extensions and revisions to our chocolate machine model. With each revision we can check the properties we have already defined, specifically those regarding post-completion errors. Each of these extended models shows how we can through experimentation and interaction with our model develop them beyond their initial design whilst retaining the properties and valuable checks we have developed.

2.3.1 Allowing different types of coins

Suppose we expand this model to allow the chocolate machine to accept both one and two pence coins. We now have two initial steps, one that represents the insertion of a 1p coin and the second for the insertion of a 2p coin:

```
pred enter2pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(2)] &&
  c'.dispenser = c.dispenser &&
  c'.state = Coin
}

pred enter1pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(1)] &&
  c'.dispenser = c.dispenser &&
  c'.state = Coin
}
```

We must also account for two different reset conditions. We are allowed to insert 2p coins into the machine for chocolates that only cost 1p and must therefore return change of 1p. However, should a 1p coin be inserted into the machine, it is not necessary to give change so we can reset immediately under these conditions. We have two reset possibilities:

```
pred reset[c,c' : Choc] {
  c.state = Change &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}

pred reset2[c,c' : Choc] {
  c.state = Chocolate &&
  c.balance = 0 &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}
```

The first allows the machine to return to the reset state after the change has been dispensed and the second allows the machine to be reset immediately after the chocolate is dispensed if and only if there is no chance to dispense.

This model is listed in appendix B.

2.3.2 Allowing different types of chocolate

Our chocolate currently only allows for the selection of one type of chocolate. Clearly, real-life chocolate machines allow for the selection of a wide variety of products. In accordance with Hoare's famous chocolate machine examples [Hoa86] we allow for the purchase of both a small chocolate, costing one

pence, and a large chocolate, costing two pence. Whereas before we had a single operation, `buychoc`, to represent the purchase of a chocolate we now introduce an operation for each of the different types of chocolate.

```
pred buysmallchoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(1)] &&
  c'.balance = INT/sub[c.balance, int(1)] &&
  no c.dispenser &&
  c'.dispenser = SmallChoc
  c'.state = Chocolate
}

pred buylargechoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(2)] &&
  c'.balance = INT/sub[c.balance, int(2)] &&
  no c.dispenser &&
  c'.dispenser = LargeChoc
  c'.state = Chocolate
}
```

Additionally, we now consider the actual retrieval of the chocolate from the dispenser. Until now, we have not considered the customer retrieving the chocolate after the machine has dispensed it. We introduce a new state, `GotChocolate`, to represent the state of the machine after the chocolate has been retrieved. (The machine may, for instance, detect this with a sensor on the dispenser door.) For all of these modifications we are introducing the actions of the customer in the form of the `OP` signature declarations and the traces model is adapted accordingly.

```
pred getchoc[c,c':Choc] {
  c.state = Chocolate &&
  c'.state = GotChocolate &&
  some c.dispenser &&
  no c'.dispenser &&
  c'.balance = c.balance
}
```

This model is listed in appendix C.

2.3.3 Purchasing more than one chocolate

Suppose the customer inserts a two pence coin but then only purchases a chocolate costing one pence. We would like to allow for multiple chocolates to be selected from the machine. A simple extension to the existing `getchoc` predicate allows there to be a valid transition from the `Chocolate` state back to the `Coin` state if a non-zero balance remains. If the customer had acted as described, entering a two pence and selecting a one pence chocolate, it would allow the machine to return to the state as if the customer had simply entered a one pence coin. This demonstrates that the state of the model can depend quite specifically on the state of the machine before the operation.

```
pred getchoc[c,c':Choc] {
  c.state = Chocolate &&
  some c.dispenser &&
  no c'.dispenser &&
  c'.balance = c.balance &&
  ( (INT/gte[c.balance,int(0)] => c'.state = Coin )
     or c'.state = GotChocolate)
}
```

This model is listed in appendix D.

2.3.4 Inserting more than one coin

In our final example, we allow the customer to insert more than one coin into the chocolate machine. This modification simply requires us to allow the machine to return to the state where it accepts coins but without resetting the stored balance.

```
pred enter2pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(2)] &&
  c'.dispenser = c.dispenser &&
  (c'.state = Coin or c'.state = Reset)
}

pred enter1pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(1)] &&
  c'.dispenser = c.dispenser &&
  (c'.state = Coin or c'.state = Reset)
}
```

This model is listed in appendix E.
We now find ourselves with a reasonably complicated chocolate machine. We began with a chocolate machine that could accept one single coin and dispense one type of chocolate. Now we have a model of a chocolate machine that can accept multiple coins of different values, provides a selection of different chocolate types and allow for the purchase of multiple chocolates. Thoroughout these modifications we have retained the ability to look for post-completion errors. The trace of the sequences of operations that results in a post-completion error has lengthened due to the extra complexity of the model.

3 Example: Cash Machine

The second example we present considers the interaction of a bank customer with a cash machine. Although a cash machine may serve many purposes, the core functionality is to dispense cash to correctly authenticated users. This process of authentication requires that the user give their cash card to the machine and correctly enter their PIN. An amount of cash is then entered followed by their card and the correct amount of cash being returned to them. A common

post-completion error with a cash machine is to collect the cash (the primary goal) but then neglect to retrieve the cash card (a sub-goal).

The following series of examples serves to illustrate how an Alloy model can develop and evolve in response to interaction and experimentation. As problems are exposed and new factors considered, the model itself can be revised and updated. It is not our intention to present a sanitised history of the models and connect them in an idealised form. We acknowledge the many dead-ends, faulty assumptions and incorrect reasoning we have made in the development of these models. We do not believe this belittles the formal development, but rather the opposite: it is unreasonable to assume we would be able to develop the Alloy models perfectly at the first attempt. Any good formal model is the result of sustained investigation into the problem domain and the equivalent effort synthesising this into a precise formal description.

3.1 Initial cash machine model

As our example we consider a simple cash machine, or automatic teller machine (ATM), whose primary function is to automatically dispense cash to customers but may provide additional functionality such as balance enquiry. Performing any financial transaction is dependent upon a customer's ability to successfully authenticate themselves to the ATM. We will model an ATM that allows a customer to both query their account balance and to withdraw cash from their account. We must model the process of authenticating a customer, by means of inserting their bank card and entering their PIN. Our model must be capable of examining the behavioural properties of the ATM to capture the interaction between customer and the user interface.

The ATM example initially declares the signature `ATMState` which will be used to represent the state of the ATM machine. Next, several other signatures are declared, all of which extend `ATMState`. Note that the `ATMState` signature is declared with the `abstract` keyword to ensure that any `ATMState` is one of `AwaitCard`, `AwaitPin`, `AwaitInst`, `RemCash` or `RemCard`.

```
abstract sig ATMstate {}
one sig AwaitCard, AwaitPin, AwaitInst,
               RemCash, RemCard extends ATMstate {}

sig ATM {
  card : lone Identifier,
  pin : lone Identifier,
  state : one ATMstate,
  balance:Identifier -> one Int,
  op:OP
}
```

The body of the signature `ATM` consists of several fields used to represent the current state of the ATM machine. For example, the first field in the ATM signature defines a relation labelled `card` which connects **ATM**s to an **Identifier**. The field `balance` is a relation connecting ATMs to Identifier to Integers. Note the presence of the multiplicity prefixes, such as `lone` and `one`, in relation declarations and used to introduce constraints onto the relations. For example, the `state` relation means that each `ATM` has exactly one `ATMstate`.

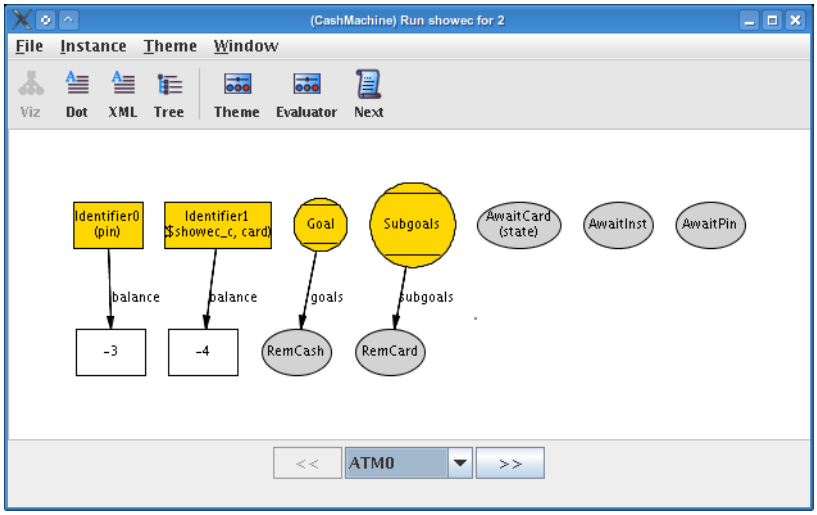

Figure 3: Example of Alloy's visualisation output for the cash machine model.

As with the chocolate machine, we use predicates to represent operations and use them to connect two possible states of the ATM machine: the before (`a`) and the after (`a'`). The predicate does not describe how to operationally transform one state into another but describes how the two are related. The model contains several operations, for example, the `entercard` predicate defines the relationship between two states: firstly when the ATM is waiting for a bank card to be inserted and then second after a bank card has been inserted and the ATM is waiting for the PIN to be entered. We define the state beforehand (`AwaitCard`) and afterwards (`AwaitPin`). The entercard predicate is as follows:

```
pred entercard [a,a':ATM, c:Identifier] {
  a.state = AwaitCard &&
  a'.card = c &&
  a'.pin = a.pin &&
  a'.balance = a.balance &&
  a'.state = AwaitPin
}
```

The structure of the signatures and the use of relations to describe relationships such as `balance` demonstrate the way that Alloy can model complex structures and general system state. In this respect, it is closer to a state-based method such as Z rather than a finite-state model-checking approach. Specifications can be written with reference to sets of infinite cardinality such as, in this case, integers.

Even at this early stage the user can ask Alloy to show some example states by executing a `run` command. This asks Alloy to generate an instance that satisfies a given constraint. For example, the following commands direct Alloy to find, if possible, an instance of the `entercard` operation. The visualisation output produced is shown in figure 3:

```
pred showec [a,a':ATM, c:Identifier] {
```

```
        entercard[a,a',c]
}
run showec for 2
```

The model can be further expanded to include operations such as entering a pin number, requesting cash, taking cash, viewing the balance and returning the card. These can be defined in a similar manner to `entercard`. Clearly, a customer's interaction with the ATM will consist of a sequence of the available operations, and this leads to the consideration of valid traces of operations from a defined initial state. We used Alloy's several library modules for the chocolate machine example and we take a similar approach here by including the *util/ordering* module, used to define a trace of operations by declaring the initial state (with the `init` predicate and `first` function), the next state (with the `next` function) and the final state (with the `last` function). This allows the dynamic behaviour of the ATM machine to be modelled by considering interactions through a series of operations. Our first attempt at the cash machine model is listed in appendix F.

3.2 Checking for post-completion errors

We consider a specific type of omission error called a post-completion error. Post-completion errors occur when a user achieves their main goal but neglects a sub-goal. In the context of our ATM example, the user may achieve the stated goal of withdrawing cash but then forget to retrieve their card, resulting in a completion error.

To capture the completion and non-completion of the goals and sub-goals, we must first introduce the idea of a transaction. We define a transaction to be a portion of a trace that begins with a user entering their card into the machine and ending with the removal of the card. Transactions are defined as follows:

```
pred transaction {
        traces &&
        (AO/last[]).op = OUTCARD &&
        OUTCARD !in (ATM - AO/last[]).op
}
```

OUTCARD represents the operation of the user removing their card from the machine, so we define this to be the last operation in the trace and prevent it from appearing any earlier in the transaction. We also declare signatures to represent our goal (retrieving the cash) and subgoal (retrieving the card). We want to ensure that all subgoals are met by the time the last primary goal is satisfied. Both the `Goal` and `Subgoals` signature are declared with appended facts, which are used to state something that must be true about the model.

```
one sig Goal {goals : set ATMstate} {goals = RemCash}
one sig Subgoals {subgoals : set ATMstate}{subgoals = RemCard}
```

The appended facts state that the primary goal is reaching the state `RemCash` and the sub-goal is reaching the state `RemCard`. We could, for example, include multiple sub-goals that might include the printing of a receipt. Checking for the post-completion property in the model uses the **assert** command, used to

state something we expect to be true as a result of our model. Examining these assertions with the **check** command directs Alloy to find a counterexample. Exhaustive search is possible for a finite state only, so the check command allows the user to specify the size of signatures to be examined. The assertion regarding goals and subgoals is as follows:

```
assert goalsmet {
  transaction => let m = AO/max[state.(Goal.goals)] |
    some m => all sg: Subgoals.subgoals | state.sg in AO/prevs[m]
}
```

This assertion states that within a transaction, all subgoals must have been achieved in states previous to that in which the primary goal is satisfied. Using the **check** command on the first version of the ATM model we are able to find the counterexample shown in figure 4. By examining the trace we can detect a problem with the dynamic behaviour of the model. The user has been able to achieve the primary goal (removing cash) before the completion of a subgoal (removing card). The automatic check reveals that a counterexample to the assertion exists within the limited scope checked. If no such counterexample had been found it is still possible that one could exist in a larger scope. The partial nature of the lightweight approach means that proof of correctness for the general case is not possible. We now explore the model further to identify the problem.

3.3 Revised cash machine model

Using Alloy to examine the first version of the model identifies a specific counterexample (as shown in figure 4). The counterexample is presented as a trace, a sequence of states that shows the primary goal is achieved before the subgoal has been satisfied. Although the Alloy Analyzer guarantees to find a counterexample if one exists, it does not guarantee to find the smallest one or the same one each time. The user is able (with practice!) to interpret the counterexample with respect to the model and identify the cause. The post-completion present in our chocolate machine example was fairly trivial but the cash machine error is slightly more complicated and not necessarily so obvious. Alloy advocates and allows for an exploratory approach towards modelling, encouraging small experiments on models to further understand and develop them. There are several strategies we can adopt to further understand the model and the counterexample. We can ask the Alloy Analyzer to find the 'next' counterexample (if one exists) which presents us with further evidence as to the problem. We may execute additional complementary checks to aid our understanding and these may take the form of more specific directed checks to narrow down on the root of the problem. The immediacy of feedback allows us to explore a possible solution in a manner difficult to achieve with full formalisation approaches.

One of the problem with interpreting the output from a traces based model is being able to visualise the progression of states. For example, the original output from the Alloy Analyzer showing the cash machine's post-completion error is shown in figure 5. This screenshot of the output shows only part of the output as it will not easily fit onto one screen. Clearly, there will be problem interpreting this output and we wish to minimise the scope for introducing

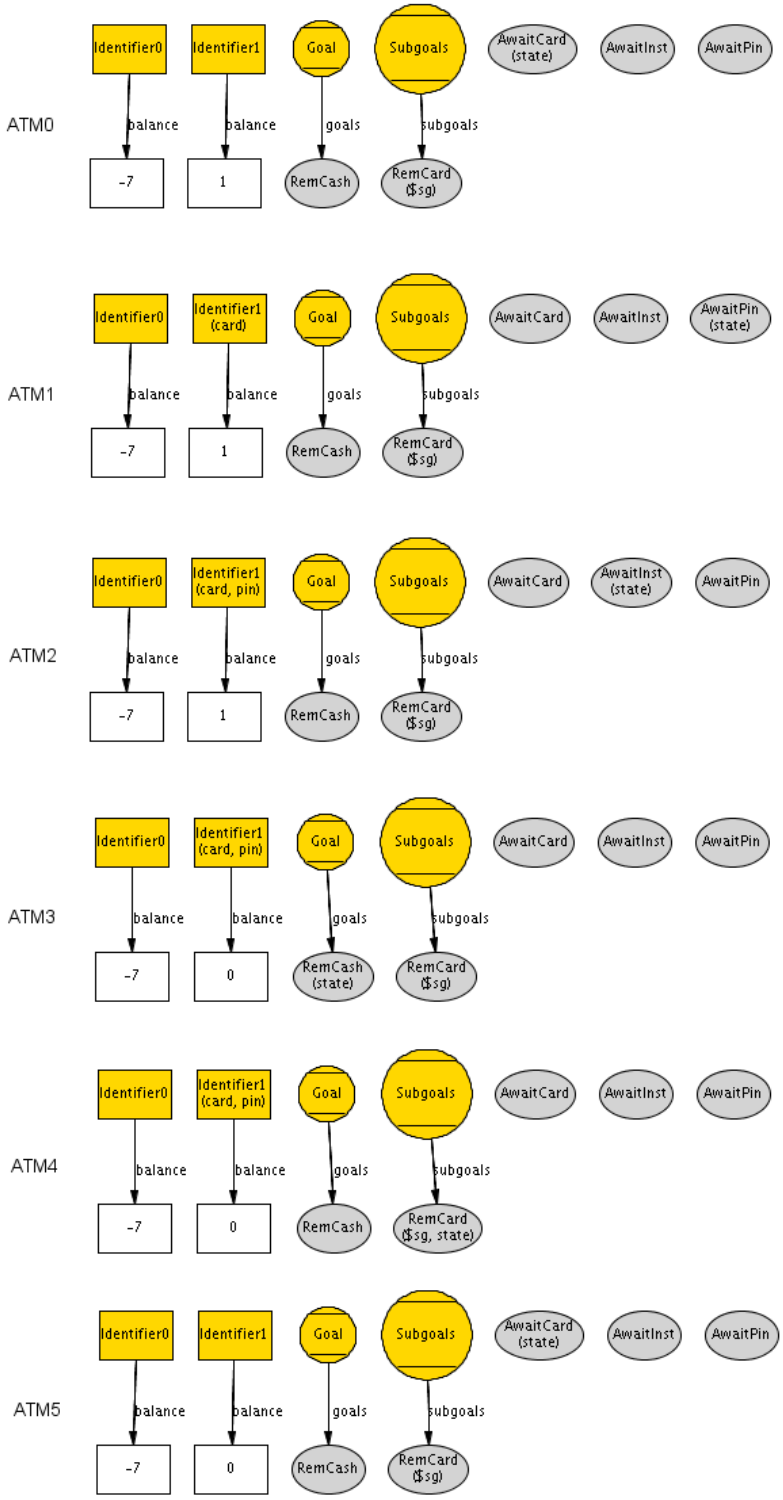

Figure 4: Counterexample trace identifying a post-completion error in the Cash-Machine model.

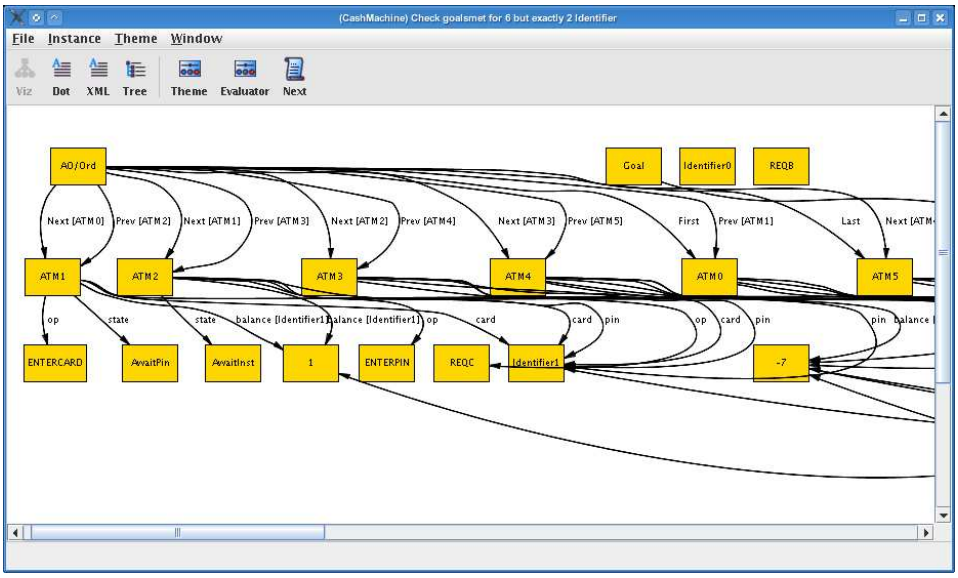

Figure 5: Unprojected trace identifying a post-completion error in the cash machine model.

additional errors. By using Alloy's projection function we can examine the state of the cash machine at each time slot. Projection over the `ATM` signature allows us to examine the sequential steps that lead from the starting state of the cash machine (shown in figure 4 as ATM0) through to the final state (shown as ATM5).

By examining the counterexample generated for the first version of the model we can see that a trace has been found where the user is allowed to remove their cash before they have removed their card. This trace is a clear violation of the stated ordering of subgoals and goals. By examining the trace we can observe that in ATM2 the current state is `AwaitInst` (indicated by the '(state)' label in the diagram). At ATM3 the state is now `RemCash` and at ATM4, `RemCard`, showing that the model incorrectly allows a valid transition (via a cash request operation) from awaiting an instruction to allowing the user to remove cash, and thence to a state in which removal of the card is expected. To rectify this, an additional relation, `pending`, is introduced into the state of the ATM to represent the money that the user has yet to remove from the dispenser. Modifying the behaviour of the `requestcash` predicate ensures that the `RemCard` state is entered into before the `RemCash` state. After introducing these changes we can immediately execute the original check to see if Alloy can now find a counterexample. After introducing these additional elements, successful execution of the previous checks indicates that the modified model has no post-completion error within the scope of the analysis. This version of the cash machine model, where the post-completion error has been fixed, is listed in appendix G.

3.4 Further revisions of the cash machine model

Two further extensions of the original model of the cash machine were developed. These extended versions of the model are listed in appendices H and I. The first extension removes the requirement for the card to be entered before the PIN, allowing either to be entered first. This is a relatively simple extension to the possible dynamic behaviour of the model with, for example, the `entercard` predicate modified as follows:

```
pred entercard [a,a':ATM, c:Identifier] {
        (a.state = AwaitCard or a.state = StartTrans) &&
        a'.card = c &&
        a'.pin = a.pin &&
        nocashchange[a,a'] &&
        (a.state = AwaitCard =>
              ((a.pin = c && a'.state = AwaitInst) or
               (a.pin != c && a'.state = RemCard))) &&
        (a.state = StartTrans => a'.state = AwaitPin)
       }
```

Whereas the original predicate only allowed the initial state to be `AwaitCard`, we now admit an additional valid initial state, `StartTrans`, which represents a transaction that will begin with the user entering their PIN rather than their card. The constraints on valid next states must also be modified. Depending on whether the user has entered their card or their pin first, the predicate to constrain possible next states is altered to allow `AwaitInst` or `RemCard`, and `AwaitPin` respectively. After these modifications, we can now re-execute the original post-completion check, to ensure we have not accidentally reintroduced the error. If were checking for many different types of interaction error, we can build up incrementally an array of checks to be performed on a model. These checks can then be run automatically at any time, allowing the model's developer to perform either small experiments or introduce larger developments whilst always retaining the ability to check their model quickly and easily. This is similar to the unit testing approach found in software development. In unit testing, when have found there a bug or believe there is the possiblity of a bug, we write a small test to check for that problem. These tests can then be executed repeatedly against our piece of software to ensure that old and often long forgotten bugs are not accidentally reintroduced by some later revision of the software.

Our fourth and final version of the ATM model is another evolutionary step in the development of the model. Good user interface practice dictates that the user should always be forgiven and allowed to reserve their actions. We therefore introduce the ability for the user to cancel their transaction at any time. This functionality requires the addition of a new predicate, `cancel`, used to represent a transaction being cancelled. This predicate allows the ATM to return to its initial state provided that we are not already waiting for the user to remove their cash from the dispenser and that the card is not present. If there is cash pending in the dispenser, the customer is required to remove their cash first. Similarly, if the customer's card remains in the slot they are required to remove their card from the machine. This predicate is as follows:

```
pred cancel [aa,aa':ATM] {
        noidchange[aa,aa'] && nocashchange[aa,aa'] &&
        (  (some aa.pending && aa'.state = RemCash ) or
           (no aa.pending && some aa.card && aa'.state = RemCard) or
           (no aa.pending && no aa.card  && init[aa'])
        )
      }
```

To account for the new predicate and new CANCEL state we must incorporate changes into the traces model. Our traces model has expanded to include the many different operations possible and includes the corresponding user operations, used to illustrate the examples generated by the tool.

Again, we are always able to refer back to the original post-completion error checks to ensure we have not reintroduced a problem. The immediacy of feedback with this approach to modelling allows us quickly to explore avenues of development for a model and also to understand the consequences of our changes. The process of analysing the model is not a protracted one and can be performed at any stage making it ideal for models which will undergo repeated enhancement.

References

[Bla00] P. Curzon & A. Blandford. Using a verification system to reason about post-completion errors. In P. Palanque & F. Patern'o, editor, *Participants Proc. of DSV-IS 2000: 7th Int.Workshop on Design, Specification and Verification of Interactive Systems, at the 22nd Int. Conf. on Software Engineerings*, pages 292–308, 2000.

[Hoa86] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1986.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.

[JW96] D. Jackson and J. M. Wing. Lightweight formal methods. *IEEE Computer*, April 1996.

[Spi89] J. M. Spivey. *The Z notation : a reference manual*. Prentice-Hall, 2nd edition, 1989.

Appendices

A Simple Chocolate Machine

```
module SimpleChocMachine

open util/ordering[Choc] as CO
open util/integer as INT

abstract sig ChocState {}
one sig Reset, Coin, Chocolate, Change extends ChocState {}

abstract sig OP {}
one sig ENTERCOIN, PUSHCHOC, PUSHCHANGE, RESET extends OP {}

abstract sig ChocType {}
one sig Choc1 extends ChocType {}

sig Choc {
  balance: one Int,
  state: one ChocState,
  op: OP,
  dispenser: lone ChocType
}

// Give us a quick example
pred show1 [c: Choc] {}
run show1

pred entercoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(2)] &&
  c'.dispenser = c.dispenser &&
  c'.state = Coin
}

pred buychoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(1)] &&
  c'.balance = INT/sub[c.balance, int(1)] &&
  no c.dispenser &&
  c'.dispenser = Choc1
  c'.state = Chocolate
}

pred askchange[c,c': Choc] {
  c.state = Chocolate
  INT/gt[c.balance, 0] &&
  INT/zero[c'.balance] &&
  c'.dispenser = c.dispenser &&
  c'.state = Change
}
```

```
pred reset[c,c' : Choc] {
  c.state = Change &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}


pred init [c:Choc] {
  c.balance = 0 &&
  c.state = Reset &&
  c.dispenser = none
}


pred traces {
        init[CO/first[]] &&
        all c:Choc-CO/last[] | let c' = CO/next[c] |
                    ((entercoin[c,c']  && c'.op = ENTERCOIN)
                     or
                     (buychoc[c,c']  && c'.op = PUSHCHOC)
                     or
                     (askchange[c,c']  && c'.op = PUSHCHANGE)
                     or
                     (reset[c,c']  && c'.op = RESET))
}

pred transaction {
        traces &&
        (CO/last[]).op = RESET &&
       RESET !in (Choc - CO/last[]).op
}

one sig Goal {goals : set ChocState} {goals = Chocolate}
one sig Subgoals {subgoals : set ChocState}{subgoals = Change}

assert goalsmet {
  transaction => let m = CO/max[state.(Goal.goals)] |
    some m => all sg: Subgoals.subgoals | state.sg in CO/prevs[m]
}

// This generates a counterexample with 5
check goalsmet for 5
```

B Chocolate machine - different coins

```
module SimpleChocMachine

open util/ordering[Choc] as CO
open util/integer as INT

abstract sig ChocState {}
one sig Reset, Coin, Chocolate, Change extends ChocState {}

abstract sig OP {}
one sig ENTER1PCOIN, ENTER2PCOIN, PUSHCHOC, PUSHCHANGE, RESET extends OP {}

abstract sig ChocType {}
one sig Choc1 extends ChocType {}

sig Choc {
  balance: one Int,
  state: one ChocState,
  op: OP,
  dispenser: lone ChocType
}

// Give us a quick example
pred show1 [c: Choc] {}
run show1

pred enter2pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(2)] &&
  c'.dispenser = c.dispenser &&
  c'.state = Coin
}

pred enter1pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(1)] &&
  c'.dispenser = c.dispenser &&
  c'.state = Coin
}

pred buychoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(1)] &&
  c'.balance = INT/sub[c.balance, int(1)] &&
  no c.dispenser &&
  c'.dispenser = Choc1
  c'.state = Chocolate
}

pred askchange[c,c': Choc] {
  c.state = Chocolate
  INT/gt[c.balance, 0] &&
  INT/zero[c'.balance] &&
```

```
  c'.dispenser = c.dispenser &&
  c'.state = Change
}

pred reset[c,c' : Choc] {
  c.state = Change &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}

pred reset2[c,c' : Choc] {
  c.state = Chocolate &&
  c.balance = 0 &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}

pred init [c:Choc] {
  c.balance = 0 &&
  c.state = Reset &&
  c.dispenser = none
}


pred traces {
        init[CO/first[]] &&
        all c:Choc-CO/last[] | let c' = CO/next[c] |
                    ((enter1pcoin[c,c']  && c'.op = ENTER1PCOIN)
                     or
                     (enter2pcoin[c,c']  && c'.op = ENTER2PCOIN)
                     or
                     (buychoc[c,c']  && c'.op = PUSHCHOC)
                     or
                     (askchange[c,c']  && c'.op = PUSHCHANGE)
                     or
                     (reset2[c,c']  && c'.op = RESET)
                     or
                     (reset[c,c']  && c'.op = RESET))
}


pred transaction {
        traces &&
        (CO/last[]).op = RESET &&
       RESET !in (Choc - CO/last[]).op
}

one sig Goal {goals : set ChocState} {goals = Chocolate}
one sig Subgoals {subgoals : set ChocState}{subgoals = Change}

assert goalsmet {
  transaction => let m = CO/max[state.(Goal.goals)] |
```

```
    some m => all sg: Subgoals.subgoals | state.sg in CO/prevs[m]
}

// This generates a counterexample with 5
check goalsmet for 5
```

C Chocolate machine - different chocolates

```
module SimpleChocMachine

open util/ordering[Choc] as CO
open util/integer as INT

abstract sig ChocState {}
one sig Reset, Coin, Chocolate, GotChocolate, Change extends ChocState {}

abstract sig OP {}
one sig ENTER1PCOIN, ENTER2PCOIN, PUSHSMALLCHOC, PUSHLARGECHOC,
   GETCHOC, PUSHCHANGE, RESET extends OP {}

abstract sig ChocType {}
one sig SmallChoc, LargeChoc extends ChocType {}

sig Choc {
  balance: one Int,
  state: one ChocState,
  op: OP,
  dispenser: lone ChocType
}

// Give us a quick example
pred show1 [c: Choc] {}
run show1

pred enter2pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(2)] &&
  c'.dispenser = c.dispenser &&
  c'.state = Coin
}

pred enter1pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(1)] &&
  c'.dispenser = c.dispenser &&
  c'.state = Coin
}

pred buysmallchoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(1)] &&
  c'.balance = INT/sub[c.balance, int(1)] &&
  no c.dispenser &&
  c'.dispenser = SmallChoc
  c'.state = Chocolate
}

pred buylargechoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(2)] &&
```

```
  c'.balance = INT/sub[c.balance, int(2)] &&
  no c.dispenser &&
  c'.dispenser = LargeChoc
  c'.state = Chocolate
}

pred getchoc[c,c':Choc] {
  c.state = Chocolate &&
  c'.state = GotChocolate &&
  some c.dispenser &&
  no c'.dispenser &&
  c'.balance = c.balance
}

pred askchange[c,c': Choc] {
  c.state = GotChocolate
  INT/gt[c.balance, 0] &&
  INT/zero[c'.balance] &&
  c'.dispenser = c.dispenser &&
  c'.state = Change
}

pred reset[c,c' : Choc] {
  c.state = Change &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}

pred reset2[c,c' : Choc] {
  c.state = Chocolate &&
  c.balance = 0 &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}

pred init [c:Choc] {
  c.balance = 0 &&
  c.state = Reset &&
  c.dispenser = none
}


pred traces {
        init[CO/first[]] &&
        all c:Choc-CO/last[] | let c' = CO/next[c] |
                  ((enter1pcoin[c,c']  && c'.op = ENTER1PCOIN)
                   or
                   (enter2pcoin[c,c']  && c'.op = ENTER2PCOIN)
                   or
                   (buysmallchoc[c,c']  && c'.op = PUSHSMALLCHOC)
                   or
                   (buylargechoc[c,c']  && c'.op = PUSHLARGECHOC)
```

```
                    or
                    (getchoc[c,c'] && c'.op = GETCHOC)
                    or
                    (askchange[c,c'] && c'.op = PUSHCHANGE)
                    or
                    (reset2[c,c'] && c'.op = RESET)
                    or
                    (reset[c,c'] && c'.op = RESET))
}


pred transaction {
        traces &&
        (CO/last[]).op = RESET &&
       RESET !in (Choc - CO/last[]).op
}

one sig Goal {goals : set ChocState} {goals = Chocolate}
one sig Subgoals {subgoals : set ChocState}{subgoals = Change}

assert goalsmet {
  transaction => let m = CO/max[state.(Goal.goals)] |
    some m => all sg: Subgoals.subgoals | state.sg in CO/prevs[m]
}

// This generates a counterexample with 6
check goalsmet for 6
```

D Chocolate machine - multiple chocolates

```
module SimpleChocMachine

open util/ordering[Choc] as CO
open util/integer as INT

abstract sig ChocState {}
one sig Reset, Coin, Chocolate, GotChocolate, Change extends ChocState {}

abstract sig OP {}
one sig ENTER1PCOIN, ENTER2PCOIN, PUSHSMALLCHOC, PUSHLARGECHOC,
  GETCHOC, PUSHCHANGE, RESET extends OP {}

abstract sig ChocType {}
one sig SmallChoc, LargeChoc extends ChocType {}

sig Choc {
  balance: one Int,
  state: one ChocState,
  op: OP,
  dispenser: lone ChocType
}

// Give us a quick example
pred show1 [c: Choc] {}
run show1

pred enter2pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(2)] &&
  c'.dispenser = c.dispenser &&
  c'.state = Coin
}

pred enter1pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(1)] &&
  c'.dispenser = c.dispenser &&
  c'.state = Coin
}

pred buysmallchoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(1)] &&
  c'.balance = INT/sub[c.balance, int(1)] &&
  no c.dispenser &&
  c'.dispenser = SmallChoc
  c'.state = Chocolate
}

pred buylargechoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(2)] &&
```

```
  c'.balance = INT/sub[c.balance, int(2)] &&
  no c.dispenser &&
  c'.dispenser = LargeChoc
  c'.state = Chocolate
}

pred getchoc[c,c':Choc] {
  c.state = Chocolate &&
  some c.dispenser &&
  no c'.dispenser &&
  c'.balance = c.balance &&
  ((INT/gte[c.balance,int(0)] => c'.state = Coin) or c'.state = GotChocolate)
}

pred askchange[c,c': Choc] {
  c.state = GotChocolate
  INT/gt[c.balance, 0] &&
  INT/zero[c'.balance] &&
  c'.dispenser = c.dispenser &&
  c'.state = Change
}

pred reset[c,c' : Choc] {
  c.state = Change &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}

pred reset2[c,c' : Choc] {
  c.state = Chocolate &&
  c.balance = 0 &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}

pred init [c:Choc] {
  c.balance = 0 &&
  c.state = Reset &&
  c.dispenser = none
}


pred traces {
        init[CO/first[]] &&
        all c:Choc-CO/last[] | let c' = CO/next[c] |
                  ((enter1pcoin[c,c']  && c'.op = ENTER1PCOIN)
                   or
                   (enter2pcoin[c,c']  && c'.op = ENTER2PCOIN)
                   or
                   (buysmallchoc[c,c']  && c'.op = PUSHSMALLCHOC)
                   or
                   (buylargechoc[c,c']  && c'.op = PUSHLARGECHOC)
```

```
                    or
                    (getchoc[c,c'] && c'.op = GETCHOC)
                    or
                    (askchange[c,c'] && c'.op = PUSHCHANGE)
                    or
                    (reset2[c,c'] && c'.op = RESET)
                    or
                    (reset[c,c'] && c'.op = RESET))
}


pred transaction {
        traces &&
        (CO/last[]).op = RESET &&
       RESET !in (Choc - CO/last[]).op
}

one sig Goal {goals : set ChocState} {goals = Chocolate}
one sig Subgoals {subgoals : set ChocState}{subgoals = Change}

assert goalsmet {
  transaction => let m = CO/max[state.(Goal.goals)] |
    some m => all sg: Subgoals.subgoals | state.sg in CO/prevs[m]
}

// This generates a counterexample with 6
check goalsmet for 6
```

E Chocolate machine - multiple coins

```
module SimpleChocMachine

open util/ordering[Choc] as CO
open util/integer as INT

abstract sig ChocState {}
one sig Reset, Coin, Chocolate, GotChocolate, Change extends ChocState {}

abstract sig OP {}
one sig ENTER1PCOIN, ENTER2PCOIN, PUSHSMALLCHOC, PUSHLARGECHOC,
  GETCHOC, PUSHCHANGE, RESET extends OP {}

abstract sig ChocType {}
one sig SmallChoc, LargeChoc extends ChocType {}

sig Choc {
  balance: one Int,
  state: one ChocState,
  op: OP,
  dispenser: lone ChocType
}

// Give us a quick example
pred show1 [c: Choc] {}
run show1

pred enter2pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(2)] &&
  c'.dispenser = c.dispenser &&
  (c'.state = Coin or c'.state = Reset)
}

pred enter1pcoin[c, c': Choc] {
  c.state = Reset &&
  c'.balance = INT/add[c.balance, int(1)] &&
  c'.dispenser = c.dispenser &&
  (c'.state = Coin or c'.state = Reset)
}

pred buysmallchoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(1)] &&
  c'.balance = INT/sub[c.balance, int(1)] &&
  no c.dispenser &&
  c'.dispenser = SmallChoc
  c'.state = Chocolate
}

pred buylargechoc[c, c': Choc] {
  c.state = Coin &&
  INT/gte[c.balance,int(2)] &&
```

```
  c'.balance = INT/sub[c.balance, int(2)] &&
  no c.dispenser &&
  c'.dispenser = LargeChoc
  c'.state = Chocolate
}

pred getchoc[c,c':Choc] {
  c.state = Chocolate &&
  some c.dispenser &&
  no c'.dispenser &&
  c'.balance = c.balance &&
  ((INT/gte[c.balance,int(0)] => c'.state = Coin) or c'.state = GotChocolate)
}

pred askchange[c,c': Choc] {
  c.state = GotChocolate
  INT/gt[c.balance, 0] &&
  INT/zero[c'.balance] &&
  c'.dispenser = c.dispenser &&
  c'.state = Change
}

pred reset[c,c' : Choc] {
  c.state = Change &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}

pred reset2[c,c' : Choc] {
  c.state = Chocolate &&
  c.balance = 0 &&
  c'.balance = c.balance &&
  c'.dispenser = c.dispenser &&
  c'.state = Reset
}

pred init [c:Choc] {
  c.balance = 0 &&
  c.state = Reset &&
  c.dispenser = none
}


pred traces {
        init[CO/first[]] &&
        all c:Choc-CO/last[] | let c' = CO/next[c] |
                  ((enter1pcoin[c,c']  && c'.op = ENTER1PCOIN)
                   or
                   (enter2pcoin[c,c']  && c'.op = ENTER2PCOIN)
                   or
                   (buysmallchoc[c,c']  && c'.op = PUSHSMALLCHOC)
                   or
                   (buylargechoc[c,c']  && c'.op = PUSHLARGECHOC)
```

```
                    or
                    (getchoc[c,c'] && c'.op = GETCHOC)
                    or
                    (askchange[c,c'] && c'.op = PUSHCHANGE)
                    or
                    (reset2[c,c'] && c'.op = RESET)
                    or
                    (reset[c,c'] && c'.op = RESET))
}


pred transaction {
        traces &&
        (CO/last[]).op = RESET &&
       RESET !in (Choc - CO/last[]).op
}

one sig Goal {goals : set ChocState} {goals = Chocolate}
one sig Subgoals {subgoals : set ChocState}{subgoals = Change}

assert goalsmet {
  transaction => let m = CO/max[state.(Goal.goals)] |
    some m => all sg: Subgoals.subgoals | state.sg in CO/prevs[m]
}

// This generates a counterexample with 6
check goalsmet for 7
```

F CashMachine.als

```
module CashMachine

open util/ordering[ATM] as AO
open util/integer as INT

sig Identifier {}

abstract sig ATMstate {}

one sig AwaitCard, AwaitPin, AwaitInst, RemCash, RemCard extends ATMstate {}

// The following definition, along with the op field of ATM, are included
// simply in order to annotate the traces when investigating system properties.

abstract sig OP {}
one sig ENTERCARD, ENTERPIN,OUTCARD, REQC, REQB, CASH extends OP {}

sig ATM { card : lone Identifier,
              pin : lone Identifier,
              state : one ATMstate,
              balance:Identifier -> one Int,
              op:OP
            }

//pred show1 [a:ATM] {some a.card}
//pred show2 [a:ATM] {some a.card && some a.pin && a.card != a.pin}
//run show2

/* Even at this early stage we can quickly get Alloy to show some example
 * states. This informs the design as we go along. Adding constraints allows
 * us to see particular instances (eg with the 2nd version of pred).
 */

pred entercard [a,a':ATM, c:Identifier] {
        a.state = AwaitCard &&
        a'.card = c &&
        a'.pin = a.pin &&
        a'.balance = a.balance &&
        a'.state = AwaitPin
       }

pred showec [a,a':ATM, c:Identifier] {
       entercard[a,a',c]
      }
run showec for 2

/* Can consider the effect of each operation as we go along. */

pred enterpin [a,a':ATM, p:Identifier] {
        a.state = AwaitPin &&
        a'.pin = p &&
        a'.card = a.card &&
```

```
        a'.balance = a.balance &&
        ((a.card = p && a'.state = AwaitInst)
         or (a.card != p && a'.state = RemCard))
       }

pred showep [a,a':ATM, c:Identifier] {
      enterpin[a,a',c]
     }
//run showep

pred outcard [a,a':ATM, c':Identifier] {
        a.state = RemCard &&
        c' = a.card &&
        a'.card = none &&
        a'.pin = none &&
        a'.balance = a.balance &&
        a'.state = AwaitCard
       }

/* Can be used in ops where the card/pin numbers stay the same */
pred noidchange [a,a':ATM]  {
        a'.card = a.card &&
        a'.pin = a.pin
       }

/* This version only does one task and then throws out the card */

pred requestcash [a,a':ATM,amount:Int]  {
        a.state = AwaitInst &&
        INT/gte[int(amount),0] &&
        noidchange[a,a'] &&
        (INT/gte[int((a.pin).(a.balance)),int(amount)]
             =>   (a'.balance = a.balance ++ a.pin ->
                      INT/sub[int((a.pin).(a.balance)),int(amount)] &&
                        a'.state = RemCash)
              else  (a'.balance = a.balance &&
                        a'.state = RemCard)
        )
       }

pred takecash [a,a':ATM] {
        a.state = RemCash &&
        noidchange[a,a'] &&
        a'.balance = a.balance &&
        a'.state = RemCard
       }

pred seebalance [a,a':ATM,b:Int] {
        a.state = AwaitInst &&
        b=(a.pin).(a.balance) &&
        noidchange[a,a'] &&
        a'.balance = a.balance &&
        a'.state = RemCard
       }
```

```
pred init [a:ATM] {
        a.card = none && a.pin = none && a.state = AwaitCard
       }

pred traces {
        init[AO/first[]] &&
        all a:ATM-AO/last[] | let a' = AO/next[a] |
            some i:Identifier |
                      ((entercard[a,a',i]  && a'.op = ENTERCARD)
                       or
                       (enterpin[a,a',i] && a'.op = ENTERPIN)
                       or
                       (outcard[a,a',i] && a'.op = OUTCARD)
                       or
                       (some amt:Int | requestcash[a,a',amt] && a'.op = REQC)
                        or
                       (takecash[a,a'] && a'.op = CASH)
                       or
                       (some amt:Int | seebalance[a,a',amt] && a'.op = REQB)
                      )
        }

// Can check out the traces //
pred multiuser  {
        traces [] &&
        (some disj i1,i2:Identifier | i1 in ATM.card && i2 in ATM.card) &&
         some a:ATM | some a.card &&  a.card != a.pin
       }
//run multiuser for 7 but 2 Identifier

//Case of money being dispensed
pred trace2 {
        traces && (AO/last[]).state = RemCash }

// Can just run to see a trace, but again, can add some constraints to see a
// variety of situations. The above yielded a trace where the user enters a
// different PIN to the one on the card. I then added the check for pin=card.
// The final line of the pred looks for a trace where a request stage is
// reached at least once.
// The check below ensures that the balance can only change via a request
// from the appropriate person.

assert samebal {
  traces => (all a:ATM-AO/last[] | let a' = AO/next[a] |
    all i:Identifier |
      i.(a.balance) != i.(a'.balance) => (a.pin = i && a'.op = REQC))
}
//check samebal for 6



// Define a transaction - a portion of a trace starting with a user entering
```

```
// a card and ending when that card is next ejected.
pred transaction {
        traces &&
        (AO/last[]).op = OUTCARD &&
        OUTCARD !in (ATM - AO/last[]).op
        }

// One drawback with the approach is that it only checks traces of exactly the
// stated length. To see possible traces, can check with different sized
// state spaces.
// Possible length traces are found to be 1, 4, 5 and 6

pred cangetcash [x :set ATMstate] {
        traces && x= ATM.state && RemCash in x}

//run transaction for 7
//run cangetcash for 6

// Now check for possible "completion errors" by designating goal and subgoals
// for a particular task. For example, suppose the user's goal is to get money.
// A subgoal is to get their card back. Is it possible to achieve the first
// before the second?

one sig Goal {goals : set ATMstate} {goals = RemCash}
one sig Subgoals {subgoals : set ATMstate}{subgoals = RemCard}

// Either, no goals have been met or all subgoals have been met by the time
// the last goal
// is satisfied.

assert goalsmet {
  transaction => let m = AO/max[state.(Goal.goals)] |
    some m => all sg: Subgoals.subgoals | state.sg in AO/prevs[m]
}

// This generates a counterexample with 6
check goalsmet for 6 but exactly 2 Identifier

// Possibly check other characteristics of goals - eg, if there is a certain
// goal which  should be achieved by every possible transaction trace. Here, it
// doesn't seem particularly relevant. User won't always get cash. They might
// not even get the card back (eg in a version where the card is retained due
// to some anomaly),

// In order to correct the "completion error" the card needs to be ejected
// before the cash is given. In this situation there will be "cash pending" -
// money waiting to be delivered. To model this, I've added another component
// to the state representation.
// In order not to get confused, I'll make a new model for the next version -
// see CashMachine2
```

G CashMachine2.als

```
module CashMachine2

// This version of the CashMachine corrects the completion error by ejecting
// the card before the money is given. The money pending is recorded in
// the state.

open util/ordering[ATM] as AO
open util/integer as INT

sig Identifier {}

abstract sig ATMstate {}

one sig AwaitCard, AwaitPin, AwaitInst, RemCash, RemCard extends ATMstate {}

// The following definition, along with the op field of ATM, are included
// simply in order to annotate the traces when investigating system properties.

abstract sig OP {}
one sig ENTERCARD, ENTERPIN,OUTCARD, REQC, REQB, CASH extends OP {}

sig ATM { card : lone Identifier,
              pin : lone Identifier,
              state : one ATMstate,
              balance:Identifier -> one Int,
              pending: lone Int,
              op:lone OP
            }

//pred show1 [a:ATM] {some a.card}
//pred show2 [a:ATM] {some a.card && some a.pin && a.card != a.pin}
//run show2

/* Even at this early stage we can quickly get Alloy to show some example
 * states.  This informs the design as we go along. Adding constraints allows
 * us to see particular instances (eg with the 2nd version of pred).
 */

/* Can be used in ops where the card/pin numbers stay the same */
pred noidchange [a,a':ATM]  {
        a'.card = a.card &&
        a'.pin = a.pin
       }

/* Can be used in ops where money details stay the same */
pred nocashchange [a,a':ATM]  {
        a'.balance = a.balance &&
        a'.pending = a.pending
       }

pred entercard [a,a':ATM, c:Identifier] {
        a.state = AwaitCard &&
```

```
        a'.card = c &&
        a'.pin = a.pin &&
        nocashchange[a,a'] &&
        a'.state = AwaitPin
       }

//pred showec [a,a':ATM, c:Identifier] {
//       entercard[a,a',c]
//       }
//run showec

pred enterpin [a,a':ATM, p:Identifier] {
        a.state = AwaitPin &&
        a'.pin = p &&
        a'.card = a.card &&
        nocashchange[a,a'] &&
        ((a.card = p && a'.state = AwaitInst) or
         (a.card != p && a'.state = RemCard))
       }

pred showep [a,a':ATM, c:Identifier] {
       enterpin[a,a',c]
      }
//run showep

pred outcard [a,a':ATM, c':Identifier] {
        a.state = RemCard &&
        c' = a.card &&
        a'.card = none &&
        a'.pin = none &&
        nocashchange[a,a'] &&
        ((no a.pending && a'.state = AwaitCard) or
         (some a.pending && a'.state = RemCash))
       }

/* This version only does one task and then throws out the card */

pred requestcash [a,a':ATM,amount:Int]  { // need to consider balances etc?
        a.state = AwaitInst &&
        INT/gt[int(amount),0] &&
        noidchange[a,a'] &&
        (INT/gte[int((a.pin).(a.balance)),int(amount)]
             =>   (a'.balance = a.balance ++ a.pin ->
                       INT/sub[int((a.pin).(a.balance)),int(amount)] &&
                       a'.pending = amount &&
                       a'.state = RemCard)
              else  (nocashchange[a,a']  &&
                       a'.state = RemCard)
        )
       }

pred takecash [a,a':ATM,c':Int] {
        a.state = RemCash &&
        c'=a.pending &&
```

```
        a'.pending = none &&
        noidchange[a,a'] &&
        a'.balance = a.balance &&
        a'.state = AwaitCard
       }

pred seebalance [a,a':ATM,b:Int] {
        a.state = AwaitInst &&
        b=(a.pin).(a.balance) &&
        noidchange[a,a'] &&
        nocashchange[a,a'] &&
        a'.state = RemCard
       }

pred init [a:ATM] {
        a.card = none && a.pin = none && a.state = AwaitCard &&
        a.pending = none && a.op = none
       }

pred traces {
        init[AO/first[]] &&
        all a:ATM-AO/last[] | let a' = AO/next[a] |
            some i:Identifier |
                      ((entercard[a,a',i] && a'.op = ENTERCARD)
                       or
                       (enterpin[a,a',i] && a'.op = ENTERPIN)
                       or
                       (outcard[a,a',i] && a'.op = OUTCARD)
                       or
                       (some amt:Int | requestcash[a,a',amt] && a'.op = REQC)
                        or
                       (some amt:Int | takecash[a,a',amt] && a'.op = CASH)
                       or
                       (some amt:Int | seebalance[a,a',amt] && a'.op = REQB)
                      )
        }

// Can check out the traces //
pred multiuser  {
        traces [] &&
        (some disj i1,i2:Identifier | i1 in ATM.card && i2 in ATM.card) &&
         some a:ATM | some a.card && a.card != a.pin  &&  RemCash
in ATM.state
       }
//run multiuser for 7 but 2 Identifier

//Case of money being dispensed
pred trace2 {
        traces && (AO/last[]).state = RemCash }

//run trace2 for 4

//assert correctpin {
//     traces => all a:ATM | a.state = AwaitInst => a.pin = a.card && some a.pin
```

```
//   }
//check correctpin for 6

// Can just run to see a trace, but again, can add some constraints to see
// a variety of situations. The above yielded a trace where the user enters a
// different PIN to the one on the card. I then added the check for pin=card.
// The final line of the pred looks for a trace where a request stage is
// reached at least once. The check below ensures that the balance can only
// change via a request from the appropriate person.

assert samebal {
       traces => (all a:ATM-AO/last[] | let a' = AO/next[a] |
                  all i:Identifier |
                   i.(a.balance) != i.(a'.balance) =>
                     (a.pin = i && a'.op = REQC))
       }
//check samebal for 6

// Alter the definition of transaction - a portion of a trace starting with
// a user entering a and ending when the machine is again ready to take a
// card.
pred transaction {
        traces &&
        (AO/last[]).state = AwaitCard &&
        AwaitCard !in (ATM - (AO/last[] + AO/first[])).state
        }

// One drawback with the approach is that it only checks traces of exactly
// the stated length. To see possible traces, can check with different sized
// state spaces.  Possible length traces are found to be 1, 4, 5 and 6

pred cangetcash [x :set ATMstate] {
        transaction && x= ATM.state && RemCash in x &&
        no (Identifier.((AO/last[]).balance) & 0)}

//run transaction for 6
//run cangetcash for 6

// Now check for possible "completion errors" by designating goal and subgoals
// for a particular task. For example, suppose the user's goal is to get money.
// A subgoal is to get their card back. Is it possible to achieve the first
// before the second?

one sig Goal {goals : set ATMstate} {goals = RemCash}
one sig Subgoals {subgoals : set ATMstate}{subgoals = RemCard}

// Either, no goals have been met or all subgoals have been met by the time
// the last goal is satisfied.

assert goalsmet {
        transaction => let m = AO/max[state.(Goal.goals)] |
                some m => all sg: Subgoals.subgoals | state.sg in AO/prevs[m]
       }
```

```
// Now no counterexample is found
check goalsmet for 6

// The completion error is now avoided.

// Another possible scenario (either order of entering pin or card) in
// CashMachine3
```

H CashMachine3.als

```
module CashMachine2

// This version of the CashMachine allows users to enter either pin or
// card first.

open util/ordering[ATM] as AO
open util/integer as INT

sig Identifier {}

abstract sig ATMstate {}

one sig StartTrans, AwaitCard, AwaitPin, AwaitInst,
        RemCash, RemCard extends ATMstate {}

// The following definition, along with the op field of ATM, are included
// simply in order to annotate the traces when investigating system
// properties.

abstract sig OP {}
one sig ENTERCARD, ENTERPIN,OUTCARD, REQC, REQB, CASH extends OP {}

sig ATM { card : lone Identifier,
              pin : lone Identifier,
              state : one ATMstate,
              balance:Identifier -> one Int,
              pending: lone Int,
              op:lone OP
            }

//pred show1 [a:ATM] {some a.card}
//pred show2 [a:ATM] {some a.card && some a.pin && a.card != a.pin}
//run show2

/* Even at this early stage we can quickly get Alloy to show some example
 * states.  This informs the design as we go along. Adding constraints allows
 * us to see particular instances (eg with the 2nd version of pred).
 */

/* Can be used in ops where the card/pin numbers stay the same */
pred noidchange [a,a':ATM]  {
        a'.card = a.card &&
        a'.pin = a.pin
       }

/* Can be used in ops where money details stay the same */
pred nocashchange [a,a':ATM]  {
        a'.balance = a.balance &&
        a'.pending = a.pending
       }

pred entercard [a,a':ATM, c:Identifier] {
```

```
        (a.state = AwaitCard or a.state = StartTrans) &&
        a'.card = c &&
        a'.pin = a.pin &&
        nocashchange[a,a'] &&
        (a.state = AwaitCard =>
         ((a.pin = c && a'.state = AwaitInst) or
          (a.pin != c && a'.state = RemCard))) &&
        (a.state = StartTrans => a'.state = AwaitPin)
       }

//pred showec [a,a':ATM, c:Identifier] {
//       entercard[a,a',c]
//       }
//run showec

pred enterpin [a,a':ATM, p:Identifier] {
        (a.state = AwaitPin or a.state = StartTrans) &&
        a'.pin = p &&
        a'.card = a.card &&
        nocashchange[a,a'] &&
        (a.state = AwaitPin =>
              ((a.card = p && a'.state = AwaitInst) or
               (a.card != p && a'.state = RemCard))) &&
        (a.state = StartTrans => a'.state = AwaitCard)
       }

pred showep [a,a':ATM, c:Identifier] {
      enterpin[a,a',c]
      }
//run showep

pred outcard [a,a':ATM, c':Identifier] {
        a.state = RemCard &&
        c' = a.card &&
        a'.card = none &&
        a'.pin = none &&
        nocashchange[a,a'] &&
        ((no a.pending && a'.state = AwaitCard) or
         (some a.pending && a'.state = RemCash))
       }

/* This version only does one task and then throws out the card */

pred requestcash [a,a':ATM,amount:Int]  { // need to consider balances etc?
        a.state = AwaitInst &&
        INT/gt[int(amount),0] &&
        noidchange[a,a'] &&
        (INT/gte[int((a.pin).(a.balance)),int(amount)]
             =>   (a'.balance = a.balance ++ a.pin ->
                       INT/sub[int((a.pin).(a.balance)),int(amount)] &&
                       a'.pending = amount &&
                       a'.state = RemCard)
              else  (nocashchange[a,a']  &&
                       a'.state = RemCard)
```

```
        )
       }

pred takecash [a,a':ATM,c':Int] {
        a.state = RemCash &&
        c'=a.pending &&
        a'.pending = none &&
        noidchange[a,a'] &&
        a'.balance = a.balance &&
        a'.state = StartTrans
       }

pred seebalance [a,a':ATM,b:Int] {
        a.state = AwaitInst &&
        b=(a.pin).(a.balance) &&
        noidchange[a,a'] &&
        nocashchange[a,a'] &&
        a'.state = RemCard
       }

pred init [a:ATM] {
        a.card = none && a.pin = none && a.state = StartTrans &&
        a.pending = none && a.op = none
       }

pred traces {
        init[AO/first[]] &&
        all a:ATM-AO/last[] | let a' = AO/next[a] |
            some i:Identifier |
                     ((entercard[a,a',i] && a'.op = ENTERCARD)
                      or
                      (enterpin[a,a',i] && a'.op = ENTERPIN)
                      or
                      (outcard[a,a',i] && a'.op = OUTCARD)
                      or
                      (some amt:Int | requestcash[a,a',amt] && a'.op = REQC)
                       or
                      (some amt:Int | takecash[a,a',amt] && a'.op = CASH)
                      or
                      (some amt:Int | seebalance[a,a',amt] && a'.op = REQB)
                     )
        }

// Can check out the traces //
pred multiuser  {
        traces [] &&
        (some disj i1,i2:Identifier | i1 in ATM.card && i2 in ATM.card) &&
         some a:ATM |
           some a.card &&  a.card != a.pin  &&  RemCash  in ATM.state
       }
//run multiuser for 7 but 2 Identifier

//Case of money being dispensed
pred trace2 {
```

```
        traces && (AO/last[]).state = RemCash }

//run trace2 for 4

//assert correctpin {
//     traces => all a:ATM | a.state = AwaitInst => a.pin = a.card && some a.pin
//   }
//check correctpin for 6

// Can just run to see a trace, but again, can add some constraints to see
// a variety of situations. The above yielded a trace where the user enters
// a different PIN to the one on the card. I then added the check for pin=card.
// The final line of the pred looks for a trace where a request stage is
// reached at least once. The check below ensures that the balance can only
// change via a request from the appropriate person.

assert samebal {
       traces => (all a:ATM-AO/last[] | let a' = AO/next[a] |
         all i:Identifier | i.(a.balance) != i.(a'.balance) =>
          (a.pin = i && a'.op = REQC))
       }
//check samebal for 6

// Alter the definition of transaction - a portion of a trace starting with
// a user entering a and ending when the machine is again ready to take a card.
pred transaction {
        traces &&
        (AO/last[]).state = StartTrans &&
        StartTrans !in (ATM - (AO/last[] + AO/first[])).state
        }

// One drawback with the approach is that it only checks traces of exactly
// the stated length. To see possible traces, can check with different sized
// state spaces. Possible length traces are found to be 1, 4, 5 and 6

pred cangetcash [x :set ATMstate] {
        transaction && x= ATM.state && RemCash in x &&
        no (Identifier.((AO/last[]).balance) & 0)}

//run transaction for 6
//run cangetcash for 6

// Now check for possible "completion errors" by designating goal and subgoals
// for a particular task. For example, suppose the user's goal is to get money.
// A subgoal is to get their card back. Is it possible to achieve the first
// before the second?

one sig Goal {goals : set ATMstate} {goals = RemCash}
one sig Subgoals {subgoals : set ATMstate}{subgoals = RemCard}

// Either, no goals have been met or all subgoals have been met by the
// time the last goal is satisfied.

assert goalsmet {
```

```
        transaction => let m = AO/max[state.(Goal.goals)] |
           some m => all sg: Subgoals.subgoals | state.sg in AO/prevs[m]
      }

// This still satisfies the completion property.
check goalsmet for 6

// The completion error is now avoided.

// Another possible scenario (either order of entering pin or card) in
// CashMachine3
```

I CashMachine4.als

```
module CashMachine2

// What happens if the user is allowed to cancel the transaction at any stage?

open util/ordering[ATM] as AO
open util/integer as INT

sig Identifier {}

abstract sig ATMstate {}

one sig StartTrans, AwaitCard, AwaitPin, AwaitInst,
        RemCash, RemCard extends ATMstate {}

// The following definition, along with the op field of ATM, are included
// simply in order to annotate the traces when investigating system properties.

abstract sig OP {}
one sig ENTERCARD, ENTERPIN,OUTCARD, REQC, REQB, CASH, CANCEL extends OP {}

sig ATM { card : lone Identifier,
              pin : lone Identifier,
              state : one ATMstate,
              balance:Identifier -> one Int,
              pending: lone Int,
              op:lone OP
            }

//pred show1 [a:ATM] {some a.card}
//pred show2 [a:ATM] {some a.card && some a.pin && a.card != a.pin}
//run show2

/* Even at this early stage we can quickly get Alloy to show some example states.
 * This informs the design as we go along. Adding constraints allows us to see
 * particular instances (eg with the 2nd version of pred).
 */

/* Can be used in ops where the card/pin numbers stay the same */
pred noidchange [a,a':ATM]  {
        a'.card = a.card &&
        a'.pin = a.pin
       }

/* Can be used in ops where money details stay the same */
pred nocashchange [a,a':ATM]  {
        a'.balance = a.balance &&
        a'.pending = a.pending
       }

pred entercard [a,a':ATM, c:Identifier] {
        (a.state = AwaitCard or a.state = StartTrans) &&
        a'.card = c &&
```

```
        a'.pin = a.pin &&
        nocashchange[a,a'] &&
        (a.state = AwaitCard =>
              ((a.pin = c && a'.state = AwaitInst) or
               (a.pin != c && a'.state = RemCard))) &&
        (a.state = StartTrans => a'.state = AwaitPin)
       }

//pred showec [a,a':ATM, c:Identifier] {
//       entercard[a,a',c]
//       }
//run showec

pred enterpin [a,a':ATM, p:Identifier] {
        (a.state = AwaitPin or a.state = StartTrans) &&
        a'.pin = p &&
        a'.card = a.card &&
        nocashchange[a,a'] &&
        (a.state = AwaitPin =>
              ((a.card = p && a'.state = AwaitInst) or
               (a.card != p && a'.state = RemCard))) &&
        (a.state = StartTrans => a'.state = AwaitCard)
       }

pred showep [a,a':ATM, c:Identifier] {
      enterpin[a,a',c]
      }
//run showep

pred outcard [a,a':ATM, c':Identifier] {
        a.state = RemCard &&
        c' = a.card &&
        a'.card = none &&
        a'.pin = none &&
        nocashchange[a,a'] &&
        ((no a.pending && a'.state = AwaitCard) or
        (some a.pending && a'.state = RemCash))
       }

/* This version only does one task and then throws out the card */

pred requestcash [a,a':ATM,amount:Int]  {  // need to consider balances etc?
        a.state = AwaitInst &&
        INT/gt[int(amount),0] &&
        noidchange[a,a'] &&
        (INT/gte[int((a.pin).(a.balance)),int(amount)]
             =>    (a'.balance = a.balance ++ a.pin ->
                        INT/sub[int((a.pin).(a.balance)),int(amount)] &&
                        a'.pending = amount &&
                        a'.state = RemCard)
              else   (nocashchange[a,a']  &&
                        a'.state = RemCard)
        )
       }
```

```
pred takecash [a,a':ATM,c':Int] {
        a.state = RemCash &&
        c'=a.pending &&
        a'.pending = none &&
        noidchange[a,a'] &&
        a'.balance = a.balance &&
        a'.state = StartTrans
       }

pred seebalance [a,a':ATM,b:Int] {
        a.state = AwaitInst &&
        b=(a.pin).(a.balance) &&
        noidchange[a,a'] &&
        nocashchange[a,a']  &&
        a'.state = RemCard
       }

// If there is some money pending, this should be delivered. If the card is
// there, return it. Otherwise, start again
pred cancel [aa,aa':ATM] {
        noidchange[aa,aa'] && nocashchange[aa,aa'] &&
        (  (some aa.pending && aa'.state = RemCash ) or
           (no aa.pending && some aa.card && aa'.state = RemCard) or
           (no aa.pending && no aa.card  && init[aa'])
        )
      }

//pred cancelok [a,a':ATM] {
//        cancel[a,a'] && a.pending != a'.pending
//      }

//run cancelok for 8

pred init [a:ATM] {
       a.card = none && a.pin = none &&
       a.state = StartTrans &&
       a.pending = none
       }

pred checkinit [a:ATM] {
       a.card = none //&& a.pin = none &&
       a.state = StartTrans &&
       a.pending = none
       }

pred traces {
        init[AO/first[]] && no (AO/first[]).op
        all a:ATM-AO/last[] | let a' = AO/next[a] |
            some i:Identifier |
                     ((entercard[a,a',i]  && a'.op = ENTERCARD)
                      or
                      (enterpin[a,a',i] && a'.op = ENTERPIN)
                      or
```

```
                        (outcard[a,a',i] && a'.op = OUTCARD)
                        or
                        (some amt:Int | requestcash[a,a',amt] && a'.op = REQC)
                         or
                        (some amt:Int | takecash[a,a',amt] && a'.op = CASH)
                        or
                        (some amt:Int | seebalance[a,a',amt] && a'.op = REQB)
                        or
                        (cancel[a,a'] && a'.op = CANCEL)
                       )
        }

// Can check out the traces //
pred multiuser  {
        traces [] &&
        (some disj i1,i2:Identifier | i1 in ATM.card && i2 in ATM.card) &&
         some a:ATM |
           some a.card &&  a.card != a.pin  &&  RemCash  in ATM.state
       }
//run multiuser for 7 but 2 Identifier

//Case of money being dispensed
pred trace2 {
        traces && (AO/last[]).state = RemCash }

//run trace2 for 4

pred trace3 {
       traces and (AO/next[AO/first[]]).op = ENTERPIN and
       (AO/last[]).op = CANCEL
       }

//run trace3


//assert correctpin {
//     traces => all a:ATM | a.state = AwaitInst => a.pin = a.card && some a.pin
//   }
//check correctpin for 6

// Can just run to see a trace, but again, can add some constraints to see
// a variety of situations. The above yielded a trace where the user enters a
// different PIN to the one on the card. I then added the check for pin=card.
// The final line of the pred looks for a trace where a request stage is
// reached at least once. The check below ensures that the balance can only
// change via a request from the appropriate person.

assert samebal {
       traces => (all a:ATM-AO/last[] | let a' = AO/next[a] |
         all i:Identifier | i.(a.balance) != i.(a'.balance) =>
           (a.pin = i && a'.op = REQC))
       }
//check samebal for 6
```

```
// Alter the definition of transaction - a portion of a trace starting
// with a user entering a and ending when the machine is again ready to
// take a card.
pred transaction {
        traces &&
        (AO/last[]).state = StartTrans &&
        StartTrans !in (ATM - (AO/last[] + AO/first[])).state
        }

// Possible trace lengths change. A user could keep cancelling over and
// over again.  Of course, this is unlikely behaviour except perhaps if
// the user doesn't know what's happening or thinks the machine's gone wrong
// and is desperately trying to stop the transaction.

pred cangetcash [x :set ATMstate] {
        transaction && x= ATM.state && RemCash in x &&
        no (Identifier.((AO/last[]).balance) & 0)}

pred trans1 {
        transaction && (AO/next[AO/first[]]).op = ENTERPIN
       }

//run transaction for 8
//run cangetcash for 6

// Now check for possible "completion errors" by designating goal and subgoals
// for a particular task. For example, suppose the user's goal is to get money.
// A subgoal is to get their card back. Is it possible to achieve the first
// before the second?

one sig Goal {goals : set ATMstate} {goals = RemCash}
one sig Subgoals {subgoals : set ATMstate}{subgoals = RemCard}

// Either, no goals have been met or all subgoals have been met by the time
// the last goal is satisfied.

assert goalsmet {
        transaction => let m = AO/max[state.(Goal.goals)] |
          some m => all sg: Subgoals.subgoals | state.sg in AO/prevs[m]
       }

// This still satisfies the completion property.
check goalsmet for 6
```