A Z schema for a vending machine

Tim French.
August 21, 2006

This document presents a simple Z specification of a vending machine. This example is adapted from the example presented in [1]. This example is intended to illustrate the use of the Z schema calculus and data structures in presenting readable schemas. Also this document (or at least it’s source) demonstrates the use of Latex to typeset Z specifications. The source file for the document, Vending.tex uses the style file zed-csp.sty written by Jim Davies. To compile this document:

1. Access a computer with Latex installed on it (for example all Linux machines in the schools labs.)
2. Place Vending.tex and zed-csp.sty into the same directory.
3. At the command line type pdflatex Vending
4. A typeset pdf file should appear in the current directory.

This document demonstrates some aspects of Z notation. Comprehensive documentation of Z notation, and how it can be typeset can be found in [2] or in [3]

1 Vending Machine Specification: the state space

In the lectures we produced a specification of Vending machine given the following constraints.

- A vending machines supplies PRODUCT’s in exchange for COIN’s.

- Each coin has a value.
- the vending machine contains a number of each product and a number of each coin.
- Coins are entered one by one.
- a product can be bought if sufficient coins have been entered, the product is available and the exact change can be given.
- The machine can be maintained, so money and products can be added and removed.

It was decided we required the types:

[PRODUCT; COIN]

...
At this stage we note that the state of vending machine seems to consist of two disjoint parts: inventory, price and the first invariant are concerned only with PRODUCT; whilst float, entered, accept and the second and third invariants are concerned only with the type COIN.

We therefore decomposed the state space in the two schemas: Inv describing the products in the machine; and Money describing the coins in the machine. These are presented in the following sections.

As a final note on the state space of the machine, we have not yet described what the value of each coin is. This could be included in the schema but doing that would suggest that the denomination of a coin may change. It is reasonable to assume that this will not happen so the value of coins can be described by a global invariant value, which is specified by an axiom:

\[
\text{value} : \text{COIN} \rightarrow \mathbb{N}
\]

2 The inventory schema

The state space of inventory is as described above:

\[
\text{Inv}
\]

\[
\text{inventory} : \text{bag PRODUCT}
\]

\[
\text{price} : \text{PRODUCT} \rightarrow \mathbb{N}
\]

\[
\text{dom inventory} \subseteq \text{dom price}
\]

\[
\text{dom float} \subseteq \text{accepted}
\]

\[
\text{dom entered} \subseteq \text{accepted}
\]

We initialize the inventory in its simplest state (i.e. empty).

3

\[
\text{InitInv}
\]

\[
\text{inventory}' = \emptyset
\]

\[
\text{price}' = 0
\]

This includes the decorated schema Inv'.

The state transitions for the Inv schema are:

1. Taking a product.
2. Restocking the inventory.
3. Changing the price of a product.
4. Query the quantity of each product.
5. Query the price of each product.

Taking a product occurs when a customer buys the product¹:

\[
\text{Take} \quad \text{item} ? : \text{PRODUCT}
\]

\[
\text{item} ? \text{in inventory}
\]

\[
\text{inventory}' = \text{inventory} \uplus [\text{item} ?]
\]

\[
\text{price} = \text{price}'
\]

Occasionally the inventory will need to be restocked. In this case the inventory is topped up by a new bag of Products (an input).

\[
\text{Restock} \quad \text{newStock} ! : \text{bag PRODUCT}
\]

\[
\text{inventory}' = \text{inventory} \uplus \text{newStock}?
\]

\[
\text{price} = \text{price}'
\]

¹Notice that zed-cap-sty interprets \text{inbag} simply as “in”, rather than the square \in sign from [2]. To maintain consistency you may prefer to use \text{item} ? \in \text{dom inventory}
Note that we need to know the price of all the new stock. Implicit in $\Delta\text{Inventory}$ is the invariant that we know the price of all stock in the inventory.

To change the price of the product we need to know the product, and the new price:

\[
\begin{align*}
\text{Price} \\
\Delta\text{Inventory} \\
\text{item}?: \text{PRODUCT} \\
\text{cost}!: \mathbb{N} \\
\text{inventory}' = \text{inventory} \\
\text{price}' = \text{price} \oplus \{\text{item}?: \text{cost'}\}
\end{align*}
\]

Note that this does not require that the product is in the inventory. We use the convention that a product is known to the machine if the price is known, and available if it is in the inventory. To Restock the machine (above) we would first have to ensure that the price is known for all elements of newStock?. If we had required that the product is in the inventory before we can Price it we would find we could not add a new product to the machine without violating the invariant.

The following two queries are simply delegated to the components of inventory (noting the $\Xi$ schema requires that no state transition occurs).

\[
\begin{align*}
\Xi\text{Product} \\
\Xi\text{Inventory} \\
\text{item}?: \text{PRODUCT} \\
\text{amount}!: \mathbb{N} \\
\text{item}?' \in \text{dom price} \\
\text{amount}' = \text{inventory} \downarrow \text{item}'
\end{align*}
\]

Note we require that the item is known. This precondition could possibly be left out without greatly affecting the specification.

3 The money schema

The state space for Money is as indicated in Section 1.

\[
\begin{align*}
\text{FindPrice} \\
\Xi\text{Inventory} \\
\text{item}?: \text{PRODUCT} \\
\text{cost}!: \mathbb{N} \\
\text{item}?' \in \text{dom price} \\
\text{cost}' = \text{price}'(\text{item}?)
\end{align*}
\]

Again the money state is initialized to be as simple as possible:

\[
\begin{align*}
\text{InitMoney} \\
\text{Money} \\
\text{float}' = \emptyset \\
\text{entered}' = \emptyset \\
\text{accepted}' = \emptyset
\end{align*}
\]

Note we use Money' rather than Money or $\Delta\text{Money}$ because we consider initialization to be an operation where only the after state is interesting.

The operations for the money schema are

1. change the set of accepted coins.
2. query how much money is in the float.
3. query how much money has been entered.
4. add coins to the float.
5. take profits from the machine.
6. enter a coin.
7. pay for a product.
8. return all entered coins.

The schema \textit{AcceptCoin} adds a coin to the set of accepted coins.

\begin{align*}
\text{AcceptCoin} & \quad \Delta \text{Money} \\
& \quad \text{newCoin}?: \text{COIN} \\
& \quad \text{newCoin} \notin \text{accept} \\
& \quad \text{float} = \text{float} \\
& \quad \text{entered}' = \text{entered} \\
& \quad \text{accept}' = \text{accept} \cup \{\text{newCoin}?'\}
\end{align*}

Again, the precondition may be considered optional. While there is unlikely to be any harm in accepting an already accepted coin, there is no need to. Making the specification as conservative as possible can simplify reasoning later.

The following transitions require a function that calculates the value of a bag of coins. There is such function native to Zed, so we will have to use axioms to specify this function, \textit{sumBag} below:

\begin{align*}
\text{sumBag} : \text{bag COIN} \rightarrow \text{N} \\
\text{sumBag}(\emptyset) = 0 \\
\forall B : \text{bag COIN}; \forall c : \text{Coin}; \forall j : \text{N} \bullet \\
\text{sumBag}(B \cup \{c \rightarrow j\}) = \text{sumBag}(B) + j \times c
\end{align*}

(the definition is from Diller [1]).

The following schemas are now trivial:

\begin{align*}
\text{FloatAmount} & \quad \exists \text{Money} \\
& \quad \text{amount}! : \text{N} \\
& \quad \text{amount}! = \text{sumBag}(<\text{float}>)
\end{align*}

\begin{align*}
\text{EnteredAmount} & \quad \exists \text{Money} \\
& \quad \text{amount}! : \text{N} \\
& \quad \text{amount}! = \text{sumBag}(<\text{entered}>)
\end{align*}

It is a good idea to be able to add some coins to the float initially.

\begin{align*}
\text{AddToFloat} & \quad \Delta \text{Money} \\
& \quad \text{newFloat}? : \text{bag COIN} \\
& \quad \text{accept}' = \text{accept} \\
& \quad \text{float}' = \text{float} \cup \text{newFloat}? \\
& \quad \text{entered}' = \text{entered}
\end{align*}

To take profits we must be able to take profit:

\begin{align*}
\text{TakeProfit} & \quad \Delta \text{Money} \\
& \quad \text{profit}? : \text{bag COIN} \\
& \quad \text{accept}' = \text{accept} \\
& \quad \text{float}' = \text{float}' \cup \text{profit}? \\
& \quad \text{entered}' = \text{entered}
\end{align*}

Note that the specification is non-deterministic. It is not necessary that we take all the money out of the float (we don’t even have to take any money out).

Coins are entered into the machine one at a time.
EnterCoin

ΔMoney
coin? : COIN

accept' = accept
float = float
entered' = entered ⊕ [coin?]

Note we must explicitly state that the float and the set of accepted coins are not altered by this operation. The precondition coin? ∈ accepted is implicit in the state invariant.

To pay for a product we do not need to know the product. We only need to know it’s cost. (Also note that unlike in the tutorial change is not given automatically.

Pay

ΔMoney
cost? : N

sumBag(entered) ≥ cost?
accept' = accept
sumBag(entered') = sumBag(entered) − cost?
float ⊕ entered = float' ⊕ entered'

The precondition is that there is sufficient money in the machine and that the correct change is available. We can easily express that there is enough money in the machine (sumBag(entered) ≥ cost?, but this is actually implied by the third predicate). That there is correct change available requires a second order statement (i.e. where we quantify over a relation type, such as ∃ b : bagCOIN • sumBag(b) = cost? ∧ b ⊆ float ⊕ entered). However, this does not have to written as a precondition as the third and fourth predicate imply that such a bag must exists. Finally we do not need to specify the postcondition: sumBag(float') = sumBag(float) + cost!, since this is also implied by the third and fourth predicate along with the axiomatic definition of sumBag.

These implications should be formally verified. There is nothing fatally wrong with including redundant predicates although they may hinder readability and simplicity of proofs.

Finally to return all coins in the machine we use the following schema which has the return type of a bag of coins:

GiveChange

ΔMoney
change! : bag COIN

accept' = accept
sumBag(float') = sumBag(float)
entered' = [ ]
float ⊕ entered = float' ⊕ change!

Again sumBag(entered) = sumBag(change!) is implied by the other predicates.

Note that the Pay schema presented in the tutorial is simply the composition of Pay with GiveChange. That is

PayTute ⊆ Pay ⊙ GiveChange

This is a simpler presentation than that given in the tute, where GiveChange was defined as

GiveChangeTute ≡ [PayTute | cost? = 0] \ cost?

4 Putting it all together

Finally to put it all together we use the schema calculus to combine the Inv and Money schemas to define a complete (although relatively abstract) specification of a Vending Machine:

VendingMachine ≡ Inv ∧ Money

InitVendingMachine ≡ InitInv ∧ InitMoney

RestockVM ≡ Restock ∧ XMoney

PriceVM ≡ Price ∧ XMoney
Buying a product is a more complex operation as it involves both money and inventory states.

\[ \text{BuyVM} \equiv (\text{Take} \land \exists \text{Money}) ; \text{FindPriceVM} \gg (\text{Pay} \land \exists \text{Inv}) \]

This is probably overusing the schema calculus. The buy operation involves taking the product then finding the price and using the \text{pipe} operator to pipe the cost of the product into the pay operation. A simpler presentation is:

\[
\begin{align*}
\text{BuyVM} & \quad \text{Take} \quad \text{Pay} \\
& \quad \text{price(item?)} = \text{cost?}
\end{align*}
\]

This schema is not identical to the one above since the \text{cost?} variable has not been hidden. Note, at this stage we may also choose to make the returning of change a part of the \text{BuyVM} schema.

The remaining operations should not affect the \text{Inv} state space so we can specify operations as

\[ \text{OpVM} \equiv \text{Op} \land \exists \text{Inv} \]

for \text{Op} being any of the schemas \text{AcceptCoin}, \text{AddToFloat}, \text{FloatAmount}, \text{EnteredAmount}, \text{EnterCoin}, \text{GiveChange}.

5 Conclusion

This presents a simpler example of a Z specification, and some hints on how to typeset Z. This document is part tutorial so the English descriptions are more verbose than necessary. English descriptions should not be used as a replacement for formality. They should be short, precise and help place the schema in context.

This example will probably be expanded on in the coming weeks. Several questions to consider are:

1. How can we make system more robust? We have specified successful operations, but a complete specification should also specify unsuccessful operations (e.g. trying to buy a product when the correct change is not available).
2. How can we improve the specification and make it more realistic? Currently we have abstracted the machine so that it can contain an unbounded amount of products and coins, but this is not feasible.
3. How can we use this specification to produce a correct implementation of the system? We will examine a process of refinement in the coming weeks.
4. How can we use proof to increase our confidence in the correctness of a specification? We saw in this example how giving an incorrect precondition could make it impossible for us to ever add stock to the machine. There are some standard theorems that can be applied to any system, and can help eliminate common errors.

References