# 1 A complete *Circus* example of ATMs using bags

A *Circus* model is a sequence of paragraphs, just like in Z, but they can also declare channels and processes. We initialise the specification by giving it a name and what other specifications does it depend on.

**section** *atm_with_bags* **parents** *bag_toolkit_spivey*, *circus_toolkit*

In our example, we first have a paragraph that declares the sets *CARD* and *PIN* of valid cards and pin numbers. We use the Z notation for introducing given sets.

$[CARD, PIN]$

We next declare some channels. Requests for money are accepted by the cash machine through the channel *incard*, which also takes a pin number and an amount to be withdrawn: the inputs are triples.

***channel*** $incard : CARD \times PIN \times \mathbb{N}_1$

The amount is a positive natural number. Cards are returned through a channel *outcard*, unless there is a problem with the card and it is retained.

***channel*** $outcard : CARD$

The notes kept in and dispensed by the cash machine are those whose denominations are in the set *Note* defined below (in the standard Z notation).

$Note == \{10, 20, 50\}$

For simplicity, we consider just a few notes, and do not address the fact that the amount requested must be decomposable in terms of the notes available. If it is not, the machine fails to dispense the cash. In our model, cash is represented as a bag [**?**] of notes: elements of the the set *Note*.

$Cash == \mathrm{bag}\, Note$

If there is enough money in the machine and a way of providing the requested amount, the cash is output through a channel *cash*.

***channel*** $cash : Cash$

The cash machine has two main components: a card verifier, which accepts requests and decides whether the card should be returned and the cash dispensed, and a cash controller, which dispenses the cash if possible, and refills the note bank. These components interact through the channels below.

***channel*** $disp : \mathbb{N};\ ok$

The channel *disp* is used by the card verifier to tell the cash controller to dispense a given amount, and *ok* is used by the cash controller to tell the card verifier that it has concluded its operation. The channel *ok* does not have a type; it is not used to communicate values, but just for synchronisation. This is also the case of the channel *refill* defined below.

$$\underline{\textbf{channel}}\ \mathit{refill}$$

This channel is used to accept requests to refill the machine

A *Circus* process models a system or a component. Just like in CSP, it interacts with its environment and other processes via channels. In *Circus*, however, a process encapsulates a state defined just like in Z. Our model defines the process *CashMachine*; its only state component is a function *noteBank* that records, for each denomination, the amount of notes available.

$$\underline{\textbf{process}}\ \mathit{CashMachine} \mathrel{\widehat=} \underline{\textbf{begin}}$$

The state is defined by a schema, namely *CMState*, which declares *noteBank* as a total function. In this example, we do not have an elaborate state invariant (which is restricted to the functional property of *noteBank*).

$$\underline{\textbf{state}}\ \mathit{CMState} == [\, \mathit{noteBank} : \mathit{Note} \pfun \mathbb{N} \,]$$

Inductive definition of summation for bags of integers, could just change $\mathbb{Z}$ to *Cash* if want to make it specific.

$$\Sigma : \mathrm{bag}\ \mathbb{Z} \to \mathbb{Z}$$

$$\begin{array}{l} \Sigma\, [\![\, ]\!] = 0 \\ \forall\, x : \mathbb{Z} \bullet \Sigma\, ([\![\, x \,]\!]) = ([\![\, x \,]\!] \sharp x) \\ \forall\, b, c : \mathrm{bag}\ \mathbb{Z} \bullet \Sigma\, (b \uplus c) = \Sigma\, b + \Sigma\, c \end{array}$$

$$\mathit{LSigma} : \mathrm{bag}\ \mathbb{Z} \to \mathbb{Z}$$

$$\begin{array}{l} \mathit{LSigma}\, [\![\, ]\!] = 0 \\ \forall\, x : \mathbb{Z};\ b : \mathrm{bag}\ \mathbb{Z} \bullet \Sigma\, ([\![\, x \,]\!] \uplus b) = ([\![\, x \,]\!] \sharp x) + \mathit{LSigma}\, b \end{array}$$

The *DispenseNotes* operation that dispenses notes takes an amount $a?$ as input and produces the bag of notes $notes!$ as output; it updates the *noteBank* accordingly. It is defined using the Z notation: a schema that defines a relation on *CMState*. The declaration $\Delta \mathit{CMState}$ introduces the variables *noteBank* to represent the value of the state component before the operation, and $\mathit{noteBank}'$ to represent its value after the operation.

*DispenseNotes*

$$\begin{array}{l} \Delta \mathit{CMState} \\ a? : \mathbb{N} \\ \mathit{notes}! : \mathit{Cash} \\ \hline \Sigma\, \mathit{notes}! = a? \\ \forall\, n : \mathit{Note} \bullet (\mathit{notes}! \sharp n) \leq \mathit{noteBank}\, n \\ \qquad \land\ \mathit{noteBank}'\, n = (\mathit{noteBank}\, n) - (\mathit{notes}! \sharp n) \end{array}$$

The value of *notes*! is nondeterministically chosen: it is any bag *notes*! whose sum $\Sigma\, notes!$ of its elements is equal to $a?$, and such that, for each note denomination $n$, the number $notes! \sharp n$ of occurrences of $n$ is less than or equal to the number of notes of denomination $n$ in the bank.

If there is no such bag, we have an error: the output is the empty bag, and the state is not changed. This scenario is defined by the schema *DispenseError* below; it includes $\Xi CMState$ to declare implicitly the variables *noteBank* and *noteBank*′ and define (also implicitly) that their values are equal.

*DispenseError*

$$\Xi CMState$$
$$a? : \mathbb{N}$$
$$notes! : Cash$$

---

$$\neg\ \exists\, ns : Cash \bullet \Sigma\, ns = a? \land \forall\, n : Note \bullet (ns \sharp n) \leq noteBank\, n$$
$$notes! = [\![\ ]\!]$$

The total operation to *Dispense* cash is the schema disjunction of the operations *DispenseNotes* and *DispenseError*.

$$Dispense == DispenseNotes \lor DispenseError$$

For conciseness, we omit the definition of the operation $\Sigma$ for bags.

The function *pin* defines the pin number of each valid card. It is declared using a Z axiomatic description, but its scope is restricted to the process.

$$pin : CARD \rightarrow PIN$$

For simplicity, we assume that the pin numbers are constant.

The first component of the cash machine is the card verifier, which is defined below using CSP notation. It first accepts a request $incard?c.(pin\, c)?a$; this is an input of any card $c$, the particular pin number $pin\, c$, and any amount $a$. It then decides whether to retain the card, output the card using the channel *outcard* and no money, or ask the cash controller to dispense the requested amount. The decision is entirely nondeterministic: it is defined by factors outside of this model: status of the card, balance on the corresponding account, and so on. If the card verifier decides to ask for the cash to be dispensed, then it waits for an *ok* from the controller to indicate that it is finished (and the verifier can proceed recursively to accept a new request).

$$\begin{aligned} CardV \mathrel{\widehat=} incard?c.(pin\, c)?a \longrightarrow\ & (CardV \\ & \sqcap \\ & outcard.c \longrightarrow CardV \\ & \sqcap \\ & disp!a \longrightarrow ok \longrightarrow outcard.c \longrightarrow CardV) \end{aligned}$$

Nondeterminism here is in the pattern of interaction, and it is explicitly indicated using the CSP construct $\sqcap$ for internal choice.

The cash controller *CashC* offers the choice to *refill* the bank or *disp*ense some money. For simplicity, we assume that when the machine is refilled, it then has *cap* notes of each denomination.

$$\mid \quad cap : \mathbb{N}$$

This is a constant that reflects the capacity of the cash machine.

In the definition of *CashC*, we use an assignment to *noteBank*, instead of a data operation defined by a Z schema, to define the value of the state after a synchronisation on *refill*. This illustrates the possibility of use of programming constructs as well as abstract specifications in *Circus*. In particular, it is possible to define an executable *Circus* model.

If the cash controller receives a request $disp?a$ to dispense an amount $a$ of cash, it uses the operation *Dispense* defined previously to determine how cash is to be dispensed. To use that operation, *CashC* declares a local variable *notes*. The input variable $a$ and the local variable *notes* now in scope are associated with the input and output of *Dispense*, which then assigns an appropriate value to *notes*. If that value is not the empty bag, then the cash is dispensed using the channel *cash*, and then the message *ok* is sent (to *CardV*). If, on the other hand, the bag is empty, then it is not possible to output the amount of cash requested and the *ok* message is sent directly.

$$\begin{array}{l} CashC \mathrel{\widehat=} \\ \quad refill \longrightarrow (noteBank := \{\, 10 \mapsto cap, 20 \mapsto cap, 50 \mapsto cap \,\} \;;\; CashC) \\ \quad \Box \\ \quad disp?a \longrightarrow \underline{\mathbf{var}}\ notes : Cash \bullet \\ \qquad\qquad \big(Dispense\big) \;; \\ \qquad\qquad \big(notes \neq [\![\,]\!]\big) \;\&\; cash!notes \longrightarrow ok \longrightarrow CashC \\ \qquad\qquad \Box \\ \qquad\qquad \big(notes = [\![\,]\!]\big) \;\&\; ok \longrightarrow CashC \end{array}$$

This illustrates the free combination of specification and programming constructs, and the free combination of data operations and communications. There is no direct association between interactions and state changes.

So far, we defined just components of (the model of) *CashMachine*. In particular, the schemas *DispenseNotes*, *DispenseError*, and *Dispense*, which are data operations, and *CardV* and *CashC* are actions of *CashMachine*. As we have seen, they have access to the state of the process, and are defined using a combination of Z, CSP, and guarded command constructs. They are used in the specification below of a main (nameless) action that defines the behaviour of *CashMachine*. This is a parallel composition of the *CardV* and *CashC* components, synchronising on the channels *disp* and *ok*.

In *Circus*, to avoid conflicts in the access to variables, a parallel composition of actions defines the disjoint sets of variables to which each of the parallel actions have write access. All the actions can read the value of all the variables before the parallelism starts, but can modify only the variables in their

associated sets. In our example, $CardV$ does not update the state, and so is associated with the empty set $\{\}$ of variables. On the other hand, $CashC$ updates $noteBank$, and so it is associated with {noteBank}.

The parallel composition also defines the channels on which communication requires interaction from both parallel actions. In our example, they are $disp$ and $ok$. This means, for example, that $CardV$ can engage on communications using $incard$ and $outcard$ independently from $CashC$, and that $CashC$ can communicate on $refill$ and $cash$ independently from $CardV$. This freedom is an extra source of nondeterminism.

The channels $disp$ and $ok$ are used only for communication between the internal components $CardV$ and $CashC$ of $CashMachine$. Such communications are of no interest to the user of a cash machine, and so they are hidden. Just like in CSP, communications on hidden channels are not visible. To conclude, the main action of $CashMachine$ is as follows.

- $(CardV \llbracket \{\} \mid \{\!| disp, ok |\!\} \mid \{noteBank\} \rrbracket CashC) \setminus \{\!| disp, ok |\!\}$

***end***

The cash machine can be refilled while a request for cash is being processed, but not when cash is being actually dispensed.

Interaction with $CashMachine$ is only possible via the channels $incard$, $outcard$, $refill$, and $cash$, in the way defined by its main action above. There is no possibility of direct access to its state, which is encapsulated. In *Circus*, we can combine basic processes, which are defined as above using Z and CSP constructs; the combinators are the usual CSP operators for internal and external choice, parallelism, interleaving and so on.