

# COINS PROBLEM

## 1. PROBLEM DESCRIPTION

The objective of this problem, which has been set as a challenge in the MIP community, is: what is the minimum number of coins (with denominations 1,2,5,10,20,50) that allows to pay exactly any amount less than £1? Apt has developed two conceptual models for this problem [1, pp. 44-46], one with six, another with 600 variables.<sup>1</sup> Specifications for both variants are developed and discussed in this text.

## 2. Z MODEL

In both of the following model variants, the instance data is given as the sequence *denom* of possible denominations.

**2.1. Using existentially quantified variables.** This variant follows the principle of the ECLIPSe implementation (figure 1 on page 3) and uses a sequence *coins* of six variables, whose range denotes the number of occurrences per coin. The predicate part is straightforward, the first two expressions declare the instance data and align the lengths of the sequences, which is followed by a universally quantified expression to set the variable ranges. Each domain declaration is a function of the corresponding coin value.

### COINS

---

*coins, denom* : seq  $\mathbb{N}_1$

*denom* =  $\langle 1, 2, 5, 10, 20, 50 \rangle$

dom(*coins*) = dom(*denom*)

$\forall i : \text{dom } \textit{coins} \bullet \textit{coins}(i) \in 0 \dots ((100 \text{ div } \textit{denom}(i)) - 1)$

$\forall \textit{value} : 1 \dots 99 \bullet$

$\exists s : \text{ran } \textit{denom} \rightarrow \mathbb{N}_1 \mid (\forall c : \text{dom } s \bullet s(c) \in 0 \dots \textit{coins}(c)) \bullet \textit{value} = \Sigma(s)$

---

The last line contains the nested problem constraints. For each possible value in the range  $1 \dots 99$  we assert that it is possible to construct a function *s* from the coin values (given as ran *denom*) to amounts such that the sum of each number of coins times its denomination yields that *value*. We use a shortcut that is made possible by the data type of *s*. The  $\Sigma$  function (defined in the Appendix) sums over a bag of Integer numbers, multiplying each element with the number of times it occurs in the bag. The bag type is in principle a function from a set *X* to  $\mathbb{N}_1$ , cf. [3, p. 124], a condition which is satisfied by the declaration of *s*. Thus *s* can be interpreted as a bag of coin values where the range elements denote the multiplicities of the coins. The existentially quantified part of the above constraint restricts each range element of the function *s* to be below or equal to the corresponding number of *coins*.

To complete the specification, we need to define an acceptable solution in terms of an optimisation function. This is done separately in the following schema. The advantage of such a two-tiered approach is not only a clearer specification, it allows us to treat the above schema as a unit in terms of which we specify the solution.

---

*Date*: December 2003.

<sup>1</sup>Apt's version of the problem uses Euro instead of pounds.

---

*Coins\_Problem*

---

*objective* : *COINS*  $\rightarrow$   $\mathbb{N}$   
*solution* : *COINS*

---

*objective* =  $(\lambda c : \text{COINS} \bullet \text{sumseq}(c.\text{coins}))$   
*solution* =  $(\mu c : \text{COINS} \mid \text{objective}(c) = \min(\text{objective}(\{\text{COINS}\}))$

---

The *objective* function takes a *COINS* schema as argument. Likewise, the variable *solution* also uses the schema *COINS* as type. In specifying the *objective*, we use the *sumseq* function defined in the Appendix, which sums up the total number of all coins in a given schema binding *c*. The *solution*, finally, is defined as that schema binding of *COINS* (i.e., an assignment of values to all internal *COINS* variables such that the related constraints are satisfied) which exhibits the minimal value of the *objective* function. This is expressed using relational image of the set *COINS* through *objective*.

**2.2. Using a matrix of possible value combinations.** This variant, suggested in [1, pp. 45–46], has 600 variables but simpler constraints in that it dispenses with existentially quantified expressions. The *coins* and *denom* variables are used in the same manner as before, the new variable is *values*, a  $6 \times 99$  matrix.<sup>2</sup> The first line of the predicate block provides, as before, the instantiation of *denom*. This is followed by specifying the dimensions of *values* and the length of *coins*.

---

*COINS*

---

*coins, denom* : seq  $\mathbb{N}_1$   
*values* : mat  $\mathbb{N}_1$

---

*denom* =  $\langle 1, 2, 5, 10, 20, 50 \rangle$   
coldim *values* = #*coins* = #*denom*; rowdim *values* = 99  
 $\forall i : \text{dom } \text{coins} \bullet \text{coins}(i) \in 0 \dots ((100 \text{ div } \text{denom}(i)) - 1)$   
 $\forall i : 1 \dots 99 \bullet$   
 $i = \text{sumseq}\{c : 1 \dots 6 \bullet c \mapsto \text{values}_{i,c} * \text{denom}(c)\}$   
 $\forall i : 1 \dots 99; j : 1 \dots 6 \bullet$   
 $\text{values}_{i,j} \leq \text{coins}(j)$

---

The subsequent domain declaration for *coins* is also taken from the preceding schema. New here are the two last statements. The first asserts that each *i* of the 99 rows, when interpreted as a polynomial whose coefficients are given by *denom*, yields as sum value *i*. This is done by creating a sequence within the curly brackets that maps each *c* to the product of the corresponding column value in row *i* and the respective value of *denom*. This sequence is then summed over by the *sumseq* operation, defined in the Appendix. The last line contains a nested statement which restricts the values of *coins* to be the upper bounds of the corresponding numbers of coins in the matrix values. This ensures that each amount to pay that could be described by a certain row in *values* can also be paid by a combination of *coins*. Lastly, we do not need to introduce a new optimisation schema for this model, as the main decision variables remain the same. Such modularity and decoupling of blocks are key strengths of Z.

### 3. NOTES ON THE SOURCE CODE

To allow a comparison with the specifications, we provide source code for both problem variants. Figure 1 on the next page shows an example ECLiPSe program for the first variant, taken from [4, sec. 2.3.2]. For the second version, we reproduce OPL code from [2] in figure 2 on page 4. The resemblance to the Z specification is close and striking (correspondence: *coins*  $\mapsto$  `num`, *denom*  $\mapsto$  `value`, *values*  $\mapsto$  `sel`), yet this code appeared roughly eight months after writing the specification. A slight difference is that the denominations are not given as instance data, but are searched for (array `value` of decision variables). This could be a hint why,

---

<sup>2</sup>the matrix type `mat` is presented in the Appendix.

according to [2], the symmetry-breaking constraints at the bottom are essential for obtaining a solution in OPL at all. Adding these constraints to the Z specification is an easy exercise which we leave to the reader.

## REFERENCES

- [1] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
- [2] Jean-François Puget. CP Next Challenge: Simplicity of Use. Invited Talk at CP-04, 2004.
- [3] J. M. Spivey. *The Z Notation: A Reference Manual*. J. M. Spivey, Oriel College, Oxford OX1 4EW, second edition, 1998. First published 1992 by Prentice Hall, current version on <http://spivey.oriel.ox.ac.uk/mike/zrm/>.
- [4] Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, 1997.

---

```
:- lib(apply_macros).
:- lib(fd).

solve(PocketCoins,Min) :-
    PocketCoins=[P,Tw,Fv,Te,Twe,Ff],
    applist(range(0,99),[Min|PocketCoins]),
    Min #= P+Tw+Fv+Te+Twe+Ff,
    fromto(1,99,gencc(PocketCoins)),
    minimize(labeling(PocketCoins),Min).

gencc(PocketCoins>Total) :-
    Coins=[P1,Tw1,Fv1,Te1,Twe1,Ff1],
    applist(range(0,99),Coins),
    Total #= P1+2*Tw1+5*Fv1+10*Te1+20*Twe1+50*Ff1,
    maplist('#!=',Coins,PocketCoins).

range(Min,Max,Var) :- Var::Min..Max
```

FIGURE 1. ECLiPSe code for the coins problem (first variant)

---

```
range coin 1..6;
range change 1..100;

var int value[coin] in 1..100;
var int num[coin] in 1..100;
var int sel[change, coin] in 0..100;

minimize
    sum(i in coin) num[i]
subject to {
    forall(i in coin, s in change)
        sel[s,i] <= num[i];
    forall(s in change)
        sum(i in coin) value[i] * sel[s,i] = s;

    forall(i in coin: i <> 6) // symmetry-breaking
        value[i] * num[i] < value[i+1];
};
```

---

FIGURE 2. OPL code for the coins problem (second variant)