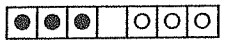
CHAPTER 6

A SOLITAIRE GAME

The solitaire game is played with three white tokens and three black ones arranged on a 7×1 board as follows:

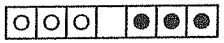


The rules of the game are:

- each white token can move to the slot to its left if it is empty, or jump over a black token to its left if the next slot is empty,
- each black token can move to the slot to its right if it is empty, or jump over a white token to its right if the next slot is empty.

Here are, two possible moves.

The aim of the game is to reach the configuration



Each slot of the board is represented by a transition system with three states: empty, black or white. An empty slot may become white by `w1` and black by `b1`. A white slot may become empty by `w0` and inform that it is white by `w`. Similarly, a black slot may become empty by `b0` and inform that it is black by `b`.

```
transition_system slot;
empty|-e ->empty,
       w1->white,
       b1->black;
white|-e- >white,
       w ->white,
       w0->empty;
black|-e ->black,
       b ->black,
       b0->empty.
```


Indeed, we need three such transition systems, with different initial states:

```
transition_system
empty;
empty|-e->empty,
      w1->white,
      b1->black;
white|-e->white,
       w->white,
       w0->empty;
black|-e->black,
       b->black,
       b0->empty;
<initial={empty}>.
```

```
transition_system
white;
empty|-e->empty,
      w1->white,
      b1->black;
white|-e->white,
       w->white,
       w0->empty;
black|-e->black,
       b->black,
       b0->empty;
<initial={white}>.
```

```
transition_system
black;
empty|-e ->empty,
       w1->white,
       b1->black;
white|-e ->white,
       w ->white,
       w0->empty;
black|-e ->black,
       b ->black,
       b0->empty;
<initial={black}>.
```

The rules of the game are defined by a synchronization constraint.

```
synchronization_system solitaire
<width=7; list=(black,black,black,empty,white,white,white)>;

\* blacks move to right *\ \* blacks jump over whites to right *\

(b0.b1.e.e.e.e.e);          (b0.w.b1.e.e.e.e);
(e.b0.b1.e.e.e.e);          (e.b0.w.b1.e.e.e);
(e.e.b0.b1.e.e.e);          (e.e.b0.w.b1.e.e);
(e.e.e.b0.b1.e.e);          (e.e.e.b0.w.b1.e);
(e.e.e.e.b0.b1.e);          (e.e.e.e.b0.w.b1);
(e.e.e.e.e.b0.b1);

\* whites move to left *\   \* whites jump over blacks to left *\

(w1.w0.e.e.e.e.e);
(e.w1.w0.e.e.e.e);          (w1.b.w0.e.e.e.e);
(e.e.w1.w0.e.e.e);          (e.w1.b.w0.e.e.e);
(e.e.e.w1.w0.e.e);          (e.e.w1.b.w0.e.e);
(e.e.e.e.w1.w0.e);          (e.e.e.w1.b.w0.e);
(e.e.e.e.e.w1.w0);          (e.e.e.e.w1.b.w0).
```

The synchronized product has 72 states and 82 transitions. Let us find the final position. At a first look it is characterized by the fact that the central slot is empty, and that it is not the initial position. Therefore we compute

```
f1:=!state[4]="empty" - initial;
```

There are 11 states in this set. Thus we look at those where the first three components are white:

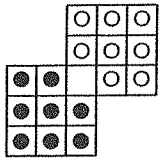
```
f2:=(f1/\!state[1]="white)/\(!state[2]="white"/\!state[3]="white");
```

As expected, there is one such state. Thus the final position is reachable. To know how many moves we need to reach it, we compute a shortest path from an initial state to a state in the set `f2`, or, more correctly, the set of transitions belonging to such a path by:

```
t:=trace(f2,*);
```

The set `t` has 15 transitions. You need 15 moves to reach the final position.

This game has several 2-dimensional versions. One of them is played on a board in the following initial configuration.



Black tokens can move upwards or to the left and white tokens can move downwards or to the right, with the same constraints as above: moving to a free neighboring slot or jumping over a neighbor token of the opposite color. The aim of the game is to reach the symmetric configuration.

The resulting transition system has 133,864 states and 229,460 transitions. The final configuration is reachable with 46 moves.

CHAPTER 7

A CAN FILLING SYSTEM

The following example is due to F. Vivares who, in her PhD thesis [41], used MEC as a verification tool for specifications obtained with Jackson's Structured Development method [25]. As we shall see there is some analogy between this method and the way we use transition systems and synchronization constraints to model a system.

7.1 The example

The system to be studied is informally described as follows:

Cans come one after the other, a robot takes them, weighs them, and sends the tare weight to the control, then pushes them to an automatic conveyor. They are carried to the filling system, which fills them; then another robot takes them, weighs them again, sends the gross weight to the control. The control verifies that the net weight is sufficient and sends the answer back to the robot. If positive, the can is put on another conveyor where it will be closed, otherwise it is put on a third conveyor leading it to recycling.

The first step is to identify the components of the system and to describe the sequential behavior of each component. In JSD this behavior is described by regular expressions, or more precisely by trees that are their parse trees. Obviously regular expressions are tightly related to transition systems.

7.1.1 Modeling behaviors of components

The systems contains three different entities: two robots and cans.

The first robot repeats the following sequence, processing one can:

- take an empty can,
- weigh it,
- deliver it to the conveyor.