Chapter 4

Blocks example

"Why," said the Dodo "the best
way to explain it is to do it"

L. Carroll. Alice's Adventures in Wonderland.

§4 Blocks example

The purpose of this chapter is illustrate how to construct a program for the abstract definitive machine whose implementation was described in Chapter 3. No new material is presented in this Chapter. In §4.4 we use mathematical techniques to show that execution of the program cannot lead to interference - this section is used to show how formal reasoning can be applied to a definitive program. The observations in §4.5 motivate the material in Chapter 5. Appendix 5 contains an adm program based on the program developed in §4.3. The problem described in this chapter was first considered in [Beynon 88].

§4.1 Specification of the problem

We wish to construct a program which can model the following system:

> "Two blocks, a and b, sit on a table which is of infinite length in either direction, and the width of one block. The position of block a is given by `pa`, that of block b by `pb`. The blocks are connected by a piece of string of length s. All quantities are integers. Each block has associated with it a handler. The handlers perform their (possibly different) actions synchronously, where an action is holding or releasing the block or, when holding the block, moving it left or right by one unit."

§4.2 Components of the model

From the specification we observe that the block handlers are independent and synchronised, a characteristic of entities within the adm. The entities in our model of the system will include blocks, handlers for the blocks, an entity to perform the block movements, and a control entity to generate the arbitrary moves of the handlers. In the course of developing the program, we will consider the boundary conditions when the need to resolve interference in actions will arise. Let d be (s+1), the maximum possible distance between the centre of the two blocks. The cases when the blocks cannot move freely and synchronously by one unit occur when:

- the string is taut (i.e. (`pb-pa`) == d)

- the string is nearly taut (i.e. (pb-pa) == d-1)
- the blocks are nearly touching (i.e. (pb-pa) == 2)
- the blocks are touching (i.e. (pb-pa) == 1)

§4.3 Program development

The blocks entity will need to store characteristics such as the position of the blocks and the distance d between the centres of the two blocks. We arbitrarily choose the initial configuration as pa=-1, pb=1 and d=5 (i.e. s=4). This is described by an entity description of the form:

```
entity blocks()
{
definition
   pa = -1, pb = 1, d = 5          // d is the distance between the
                                       // block centres
}
```

We now require handlers for the blocks. Since they are to act autonomously, we will require a separate entity instantiation for each block. We only permit a handler to push a block for one unit at a time to the left or right.

```
entity handler(block)
{
definition
   pushing[block] = pushingL[block] ⁄ pushingR[block],
   pushingL[block] = FALSE,
   pushingR[block] = FALSE,
   holding[block] = FALSE
action
   (¬holding[block])          -> holding[block] = TRUE,          (1)
   (move[block]==0) € holding[block] € ¬pushing[block]
                              -> holding[block] = FALSE,         (2)

   (move[block]==-1) € holding[block] € ¬pushing[block]
                              -> pushingL[block] = TRUE,         (3)
   pushingL[block]            -> pushingL[block] = FALSE,        (4)

   (move[block]==1) € holding[block] € ¬pushing[block]
                              -> pushingR[block] = TRUE,         (5)
   pushingR[block]            -> pushingR[block] = FALSE         (6)
}
```

We now require an entity to perform the actual movement of the blocks. This will have a skeletal outline of the form:

```
entity blockmover(blocka,blockb)
{
action
   pushingL[blocka]          -> pa = |pa|-1,
   pushingR[blocka]          -> pa = |pa|+1,
   pushingL[blockb]          -> pb = |pb|-1,
   pushingR[blockb]          -> pb = |pb|+1
}
```

Finally we require a control entity to generate the moves of the handlers. To do this we shall use the `rand()` function described in §3.5. In each execution cycle we generate new values for `move[]` which are either -1, 0 or 1. A value of -1 indicates a move to the left, a value of 0 an instruction to release, and a value of 1 a move to the right.

```
entity control(blocka,blockb)
{
definition
   move[blocka], move[blockb],
   random[blocka]=rand(3), random[blockb]=rand(3)
action
   TRUE ->    move[blocka] = |random[blocka]|-1;
              move[blockb] = |random[blockb]|-1
}
```

To configure the model the following instantiations are required:

```
blocks()
handler(0)
handler(1)
blockmover(0,1)
control(0,1)
```

We must now address the interference which occurs at the boundary conditions outlined in §4.2. We need to constrain the changes which can be made to `pa` and `pb` so as to maintain the relationship $1 \leq$ `pb-pa` $\leq d$. If the string is taut (`pb-pa`=d), then we must not allow block a to move left (i.e. for `pa` to decrease) whilst block b moves right. If the string is taut and only one block moves, then the position of the other block may need to be updated accordingly. When the distance between the two blocks is d-1 (i.e. the string is nearly taut) and the blocks are moved away from each other, the conflict must be resolved.. The converse of these three cases must also be considered when the blocks are touching (`pb-pa`=1) or nearly touching (`pb-pa`=2).

Ignoring for the moment the cases where the blocks are nearly touching or the string is nearly taut, we can enumerate the possibilities as follows:

1 pushing block a left, string taut
2 pushing block a left, string not taut
3 pushing block a right, blocks touching
4 pushing block a right, blocks not touching
5 pushing block b right, string taut
6 pushing block b right, string not taut
7 pushing block b left, blocks touching
8 pushing block b left, blocks not touching

These eight combinations give rise to corresponding guards:

```
1 pushingL[blocka] € stringtaut
2 pushingL[blocka] € ¬stringtaut
3 pushingR[blocka] € touching
4 pushingR[blocka] € ¬touching
5 pushingR[blockb] € stringtaut
6 pushingR[blockb] € ¬stringtaut
7 pushingL[blockb] € touching
8 pushingL[blockb] € ¬touching
```

If both blocks are held then we assume that block positions do not change in the odd-numbered cases, and do nothing. Otherwise we establish *contexts for action* (by means of definitions) which will cause the position of block b to change by side effect in cases 1 and 3, and the position of block a to change by side effect in cases 5 and 7. This implies a `blockmover()` entity with the following eight guards:

```
entity blockmover(blocka,blockb)
{
action
   pushingL[blocka] € stringtaut € ¬holding[blockb]
                              -> pb = pa+d; pa = |pa|-1,
   pushingL[blocka] € ¬stringtaut
                              -> pa = |pa|-1,
   pushingR[blocka] € touching € ¬holding[blockb]
                              -> pb = pa+1; pa = |pa|+1,
   pushingR[blocka] € ¬touching
                              -> pa = |pa|+1,
   pushingR[blockb] € stringtaut € ¬holding[blocka]
                              -> pa = pb-d; pb = |pb|+1
   pushingR[blockb] € ¬stringtaut
                              -> pb = |pb|+1,
   pushingL[blockb] € touching € ¬holding[blocka]
                              -> pa = pb-1; pb = |pb|-1,
   pushingL[blockb] € ¬touching
                              -> pb = |pb|-1
}
```

We have introduced two new variables, `touching` and `stringtaut`, which will be defined in the

`blocks()` entity as

```
stringtaut = ((pb-pa) == d)
touching = ((pb-pa) == 1)
```

The next situation to be addressed occurs when the blocks are nearly touching or the string is nearly taut. These respectively occur when `pb-pa`=2 and when `pb-pa`=d-1. When the string is nearly taut and the blocks are simultaneously moved apart, we shall arbitrarily allow one block to be moved faster than the other, and only change the position of that block. A more realistic modelling might change the granularity of movement to permit the two blocks to both move half a unit, but the problem specification stated that all measurements are integer. We allow a similar resolution when the blocks are nearly touching and are moved towards each other. This is specified in the following `blockmover()` and `blocks()` entity descriptions:

```
entity blockmover(blocka,blockb)
{
definition
   resolve = |rand(2)|


action
   pushingL[blocka] € stringtaut € ¬ holding[blockb]
                              -> pb = pa+d; pa = |pa|-1,
   pushingL[blocka] € neartaut €
      (¬pushingR[blockb]⁄(resolve == 0))
                              -> pa = |pa|-1,
   pushingL[blocka] € ¬stringtaut € ¬neartaut
                              -> pa = |pa|-1,

   pushingR[blocka] € touching € ¬ holding[blockb]
                              -> pb = pa+1; pa = |pa|+1,
   pushingR[blocka] € neartouch €
      (¬pushingL[blockb]⁄(resolve == 0))
                              -> pa = |pa|+1,
   pushingR[blocka] € ¬touching € ¬neartouch
                              -> pa = |pa|+1,

   pushingR[blockb] € stringtaut € ¬ holding[blocka]
                              -> pa = pb-d; pb = |pb|+1
   pushingR[blockb] € neartaut €
      (¬pushingL[blocka]⁄(resolve == 1))
                              -> pb = |pb|+1,
   pushingR[blockb] € ¬stringtaut € ¬neartaut
                              -> pb = |pb|+1,

   pushingL[blockb] € touching € ¬ holding[blocka]
                              -> pa = pb-1; pb = |pb|-1,
   pushingL[blockb] € neartouch €
```

```
      (¬pushingR[blocka] ⁄ (resolve == 1))
                              -> pb = |pb|-1,
   pushingL[blockb] € ¬touching € ¬neartouch
                              -> pb = |pb|-1,

   true                       -> resolve = |rand(2)|
}

entity blocks()
{
definition
   pa = -1, pb = 1, d = 5,
   stringtaut = ((pb-pa) == d),
   touching = ((pb-pa) == 1),
   neartaut = ((pb-pa) == d-1),
   neartouch = ((pb-pa) == 2)
}
```

We are establishing a context for an action, but then retaining that context when the conditions which led to it being established no longer hold. For example, the value of `pb` should change dependent on the value of `pa` only whilst block a is being pushed left and the string is taut. Once the condition no longer holds, the value of `pb` should not be dependent on that of `pa`. The redefinition

```
pb = |pb|
```

has the effect of removing dependencies from the definition of `pb`. It is necessary to construct a `blockmover()` entity which will maintain the appropriate dependencies for only as long as they are valid. To do this we need to make the `blockmover()` entity process more quickly than the other entities, for which purpose we introduce the notion of a global clock, and restrict the operation of `handler()` to every second "tick". This gives a final program:

```
entity handler(block)
{
definition
   pushing[block] = pushingL[block] ⁀ pushingR[block],
   pushingL[block] = FALSE,
   pushingR[block] = FALSE,
   holding[block] = FALSE

action
   on € (¬holding[block])
                              -> holding[block] = TRUE,
   on € (move[block]==0) € holding[block] € ¬pushing[block]
                              -> holding[block] = FALSE,

   on € (move[block]==-1) € holding[block] € ¬pushing[block]
                              -> pushingL[block] = TRUE,
   on € pushingL[block]       -> pushingL[block] = FALSE,

   on € (move[block]==1) € holding[block] € ¬pushing[block]
                              -> pushingR[block] = TRUE,
   on € pushingR[block]       -> pushingR[block] = FALSE
}


entity blocks()
{
definition
   pa = -1, pb = 1, d = 5,
   stringtaut = ((pb-pa) == d),
   touching = ((pb-pa) == 1),
   neartaut = ((pb-pa) == d-1),
   neartouch = ((pb-pa) == 2)
}
```

```
entity blockmover(blocka,blockb)
{
definition
   resolve = |rand(2)|

action
   on € pushingL[blocka] € stringtaut € ¬ holding[blockb]
                              -> pb = pa+d; pa = |pa|-1,
   on € pushingL[blocka] € neartaut €
      (¬pushingR[blockb]/(resolve == 1))
                              -> pa = |pa|-1,
   on € pushingL[blocka] € ¬stringtaut € ¬neartaut
                              -> pa = |pa|-1,

   on € pushingR[blocka] € touching € ¬ holding[blockb]
                              -> pb = pa+1; pa = |pa|+1,
   on € pushingR[blocka] € neartouch €
      (¬pushingL[blockb]/(resolve == 1))
                              -> pa = |pa|+1,
   on € pushingR[blocka] € ¬touching € ¬neartouch
                              -> pa = |pa|+1,

   on € pushingR[blockb] € stringtaut € ¬ holding[blocka]
                              -> pa = pb-d; pb = |pb|+1
   on € pushingR[blockb] € neartaut €
      (¬pushingL[blocka]/(resolve == 2))
                              -> pb = |pb|+1,
   on € pushingR[blockb] € ¬stringtaut € ¬neartaut
                              -> pb = |pb|+1,

   on € pushingL[blockb] € touching € ¬ holding[blocka]
                              -> pa = pb-1; pb = |pb|-1,
   on € pushingL[blockb] € neartouch €
      (¬pushingR[blocka]/(resolve == 2))
                              -> pb = |pb|-1,
   on € pushingL[blockb] € ¬touching € ¬neartouch
                              -> pb = |pb|-1,

   ¬on                        -> pa = |pa| ; pb = |pb| ;
                                 resolve = |rand(2)|
}

entity control(blocka,blockb)
{
definition
   move[blocka], move[blockb],
   random[blocka]=rand(3), random[blockb]=rand(3),
   time = 0, on = ((time div 2) == 0)

action
   TRUE ->    move[blocka] = |random[blocka]|-1;
              move[blockb] = |random[blockb]|-1;
              time = |time| + 1
}
```

The restricted nature of output means that additional methods must be used to produce animated output. A use of

DoNaLD (Definitive Notation for Line Drawing) [Beynon et al 86] for animation of a blocks program based on the program §4.3 is demonstrated in Appendix 5. It would be appropriate to develop more sophisticated methods for relating the definitive state to current output.

§4.4 Proof of correctness

We now prove some properties about the behaviour of the program. The six guards of `handler()` are:

```
(1)  on € (¬holding[block])
(2)  on € (move[block]==0) € holding[block] € ¬pushing[block]
(3)  on € (move[block]==-1) € holding[block] € ¬pushing[block]
(4)  on € pushingL[block]
(5)  on € (move[block]==1) € holding[block] € ¬pushing[block]
(6)  on € pushingR[block]
```

where the following definition always holds:

```
pushing[block] = pushingL[block] ⁄ pushingR[block],
```

Theorem 1: guards (1) to (6) in `handler()` are mutually exclusive.

Proof:

• Guard (2) is mutually exclusive with all other guards

`holding[block]` excludes guard (1).

`move[block]` = 0 excludes guards (3) and (5).

`¬pushing[block]`

- `¬(pushingL[block] ⁄ pushingR[block])`
- `¬pushingL[block] € ¬pushingR[block]  (*)`

which excludes (4) and (6).

• Guard (1) is mutually exclusive with all other guards

`¬holding[block]` excludes guards (2), (3) and (5).

The only place holding[block] is set to false is in command (2), which has been shown to be mutually exclusive with all other guards. A precondition for (2) is that ¬pushing[block] be true, so by (*) (¬pushingL[block] ∈ ¬pushingR[block]) must be true when holding[block] is set to false. Since holding[block] must be true to set pushingL[block] (guard (3)) or pushingR[block] (guard (5)) to true, this means that pushingL[block] and pushingR[block] cannot be true whenever holding[block] is false. In other words

¬holding[block] ª ¬pushing[block]

and the logically equivalent

pushing[block] ª holding[block] (**)

By (**), guard (1) is mutually exclusive with guards (4) and (6).

• Guard (3) is mutually exclusive with all other guards

move[block] = -1 excludes guard (5).

By (*), guards (4) and (6) cannot be simultaneously true with (3).

• Guard (5) is mutually exclusive with all other guards

By (*), guards (4) and (6) cannot be simultaneously true with (5).

• Guard (4) is mutually exclusive with all other guards

Because guards (3) and (5) (the ones which set pushingL[block] or pushingR[block] to true) are exclusive, pushingL[block] and pushingR[block] cannot be both set to true on the same iteration. Since a prerequisite for setting pushingL[block] to true is ¬pushing[block], pushingR[block] cannot be set to true when pushingL[block] is true. A similar argument applies for pushingR[block]. Therefore pushingL[block] and pushingR[block] cannot both be true at the same time. In other words

pushingL[block] ª ¬pushingR[block]

and

pushingR[block] ª¬pushingL[block] (***)

Therefore guards (4) and (6) are exclusive.

end of proof

From this theorem we deduce that there can never be interference in the redefinitions of holding[block],

pushingL[block] or pushingR[block]. We now consider the blockmover() entity. The guards of this entity are:

```
(1)   on ∈ pushingL[blocka] ∈ stringtaut ∈ ¬ holding[blockb]
(2)   on ∈ pushingL[blocka] ∈ neartaut
        ∈ (¬pushingR[blockb]⁄(resolve == 1))
(3)   on ∈ pushingL[blocka] ∈ ¬stringtaut ∈ ¬neartaut
(4)   on ∈ pushingR[blocka] ∈ touching ∈ ¬ holding[blockb]
(5)   on ∈ pushingR[blocka] ∈ neartouch
        ∈ (¬pushingL[blockb]⁄(resolve == 1))
(6)   on ∈ pushingR[blocka] ∈ ¬touching ∈ ¬neartouch
(7)   on ∈ pushingR[blockb] ∈ stringtaut ∈ ¬ holding[blocka]
(8)   on ∈ pushingR[blockb] ∈ neartaut
        ∈ (¬pushingL[blocka]⁄(resolve == 2))
(9)   on ∈ pushingR[blockb] ∈ ¬stringtaut ∈ ¬neartaut
(10)  on ∈ pushingL[blockb] ∈ touching ∈ ¬ holding[blocka]
(11)  on ∈ pushingL[blockb] ∈ neartouch
        ∈ (¬pushingR[blocka]⁄(resolve == 2))
(12)  on ∈ pushingL[blockb] ∈ ¬touching ∈ ¬neartouch
(13)  ¬on
```

To ensure that stringtaut, neartaut, neartouch and touching are mutually exclusive we shall assume that d≥4. This will mean that at any point during execution of the program no more than one of these four variables can be true, and there may be times when none of them will be true. From examination of the guards and the use of (**) and (***) we can deduce that the only combinations of guards which can be simultaneously true in blockmover() are:

(2,12) (3,9) (3,11) (3,12) (5,9) (6,8) (6,9) (6,12)

The commands corresponding to the relevant guards are:

```
pa = |pa|-1          (2)
pa = |pa|-1          (3)
pa = |pa|+1          (5)
pa = |pa|+1          (6)
pb = |pb|+1          (8)
pb = |pb|+1          (9)
pb = |pb|-1          (11)
pb = |pb|-1          (12)
```

This means that only the following commands can be executed simultaneously:

pa = |pa|-1 *and* pb = |pb|+1 (3,9)

`pa = |pa|-1` *and* `pb = |pb|-1`	(2,12) (3,11) (3,12)
`pa = |pa|+1` *and* `pb = |pb|+1`	(5,9) (6,8) (6,9)
`pa = |pa|+1` *and* `pb = |pb|-1`	(6,12)

If we permit d to have a value which is less than 4, then the variables `stringtaut`, `neartaut`, `neartouch` and `touching` will no longer be mutually exclusive. The analysis can be broken down as follows:

• d=3 exactly one of the variables `stringtaut`, `neartaut`, `neartouch` and `touching` must be true at any time. This does not change the combinations of guards which can simultaneously be true.

• d=2 exactly one of the expressions (`stringtaut`), (`touching`) and (`neartouch` ∈ `neartaut`) must be true at any time. In this case the only pairs of guards which can be simultaneously being true (2,11), (2,12), (5,8) and (6,12). None of these combinations involves redefinition of the same variable.

• d=1 exactly one of the expressions (`stringtaut` ∈ `neartouch`) and (`touching` ∈ `neartaut`) must be true at any time. No pairs of guards can be simultaneously true in this case.

• d=0 the expression (`stringtaut` ∈ `touching` ∈ ¬`neartaut` ∈ ¬`neartouch`) must always be true. No pairs of guards can be simultaneously true in this case.

Since no value of d allows redefinitions of the same variable, there can be no fatal error due to duplicate redefinitions. Parallel redefinitions of the same variable or redefinition of a variable not in D are the only ways in which a fatal error can be issued which do not involve dynamic actions, so there can be no fatal error issued when executing the program.

§4.5 Limitations of the adm

We have shown how to program the adm for the blocks example. The abstract definitive machine provides an environment in which relationships between components of a system can be maintained automatically, and in which parallel actions can be performed in a synchronised and analysable way. It is hoped that methods can be developed for formally analysing the behaviour of an adm program, for example in the manner of §4.4.

A number of deficiencies in the adm can be identified. The identification of these deficiencies in this section motivates the material in Chapter 5, which gives a method for separating the operational characteristics of each component of a system from consideration of the interaction and synchronisation between components.

The method of using the on variable to express relative operational speeds of entities and to cancel contexts for action is not an elegant solution to changing the context for action. The adm is well suited to the definition within appropriate states of persistent relationships. It does not easily allow transient definitions, those which replace an existing definition for a time, after which the old definition is restored.

A better method would be to supply a mechanism for specifying the context by definitions, i.e. higher order definitions. This would allow the definition associated with the variable to change as the context changes, so that sometimes it will be a literal value and at other times it will be defined as functionally dependent on other variables. As an example the definition for pb might be of the form

```
pb =  if (pushingL[blocka] ∈ stringtaut ∈ ¬ holding[blockb])
      then pa+d
      else  if (pushingR[blocka] ∈ touching ∈ ¬ holding[blockb])
            then pa+1
            else |pb|
```

where |pb| means that the value of pb is used. When the state changes in such a way as to cause the definition associated with pb to change, it is incorrect if the new view of pb changes its value. The clear delimiting of state and transition within the adm would the programmer to reason about when the change of view will occur.

A natural extension of the blocks example would involve the use of autonomous handlers which act asynchronously. This is difficult to describe within the framework of an adm program (although it can be done, as we show in Chapter 5) because of the need to represent the autonomous, unsynchronised nature of such handlers and the need to describe the arbitrary delay that there can be between a guard being selected and the corresponding command list being executed.

The "flat" definition store means that there is no information hiding possible, so modular programming is not encouraged when using the abstract defintive machine. Whilst a syntactic analysis of an entity can determine (apart from the problem of aliasing) what variables and entities it can act on, it is in principle possible for an entity to act on

any variable in D or entity in P. This makes it more complex to reason about the potential behaviour of an adm program.

In Chapter 5 we attempt to address these issues by introducing a specification notation called LSD, which uses definitions in the modelling and simulation of concurrent systems.