

THE PROBLEM OF THE LEIDSESTRAAT
an exercise in process coordination

Aard : rapport

Omvang : 21 pp.

Datum : Augustus 1981

Lab : Automatische Verkeerssystemen

Nummer : no. 36

Auteur : dr.ir. Gerard J. Holzmann

Titel : The problem of the Leidsestraat

Gerard J. Holzmann
(Delft, August 1981)

Delft University of Technology
Department of Electrical Engineering
AVS Laboratory
Mekelweg 4
Delft, 2600 GA
The Netherlands

THE PROBLEM OF THE LEIDSESTRAAT
an exercise in process coordination

THE PROBLEM OF THE LEIDSESTRAAT
an exercise in process coordination

1. Introduction

1.1 Problem Description

The Leidsestraat is a busy shopping street in Amsterdam. There is two-way traffic of trams through this street: from the Rijningsplein to the Leidseplein, and back. The street, though, is not wide enough for two separate tracks over its full length. In fact, only the two bridges in the street can accommodate two tracks; the street is just one.

ABSTRACT

This note describes an involved traffic control problem that is familiar to many people using the tramway system in Amsterdam. The problem is to find a control algorithm (i.e. a method to synchronize concurrent processes modeling tram behavior) that avoids collisions, deadlocks, and starvations of trams.

The problem is used to illustrate the application of a systematic design discipline that allows one to design the control algorithm step by step, and to prove the solution correct.

Gerard J. Holzmann
(Delft, August 1981)

The requirements for the control algorithm are that:

- * collisions are avoided
- * trams do not have to back up to resolve problems in the coordination
- * the waiting time at every traffic light in the street is always finite
- * trams are only delayed when their progress would directly lead to a violation of at least one of the above requirements.

The last requirement will ban 'solutions' that are correct, but far too strict, such as placing a traffic light at each end of the street, which would allow only one tram to drive through it at a time.

We decree that the following assumptions about the system may be made in attempts to solve the problem:

- * Trams travel at finite non-zero speeds and stop only at traffic lights.

THE PROBLEM OF THE LEIDSESTRAAT

an exercise in process coordination

1. Introduction

1.1 Problem Description

The Leidsestraat is a busy shopping street in Amsterdam. There is two-way traffic of trams through this street: from the Koningsplein to the Leidseplein, and back. The street, though, is not wide enough for two separate tracks over its full length. In fact, only the two bridges in the street can accomodate two tracks, the street itself has just one.

Clearly, the traffic through the Leidsestraat has to be coordinated to avoid head-on collisions and whatever other problems may occur (we will discover quite a few). In the real street there are no traffic lights, and coordination is achieved with a few simple priority rules (traffic from the Leidseplein goes first, the street may not be entered if the first bridge in sight is full). Unfortunately, the rules are not always observed, and even worse: they do not always work as intended. It is not an unusual sight to see trams backing up through the street in order to resolve unexpected ties.

The problem we pose is to try and find a traffic control algorithm that will ensure normal traffic through the street in both directions under all circumstances, using as many traffic lights as deemed necessary.

The requirements for the control algorithm are that:

- * collisions are avoided
- * trams do not have to back up to resolve problems in the coordination
- * the waiting time at every traffic light in the street is always finite
- * trams are only delayed when their progress would directly lead to a violation of at least one of the above requirements.

The last requirement will ban 'solutions' that are correct, but far too strict, such as placing a traffic light at each end of the street, which would allow only one tram to drive through it at a time.

We decree that the following assumptions about the system may be made in attempts to solve the problem:

- * Trams travel at finite non-zero speeds and stop only at traffic lights.

- * The arrival times of the trams at the street are not known in advance.
- * The maximum number of trams that each bridge track can hold is finite, and will be named M (in the Leidsestraat $M = 2$).
- * The number of trams that each street track can hold is finite, and will be named N. (In the real street this would be about 5 trams.)
- * The number of trams in Amsterdam is not limited.

To simplify the problem somewhat we will, furthermore, limit the version of the problem to be discussed to a segment of the street with just one bridge.

The tram control problem is used here as an example that can help us obtain a better insight into the nature of complicated process coordination problems in multiprocessing systems. It can be considered a test to our ability to solve such problems systematically and to prove that the results are really the ones intended.

We will model the trams as concurrent processes, and the tracks in the Leidsestraat as critical sections in the program code. We will attempt to derive a solution to the problem in the language Concurrent Pascal, using the monitor concept to enforce the required coordinations.

The following discussion is based upon the theoretical model first introduced in HOLZ79, and elaborated in HOLZ80. Accordingly, the design process will be divided into two main phases: (1) the derivation of a minimal and consistent coordination scheme, and (2) the design of a minimal and complete signaling scheme.

1.2 Formalization of the problem

A first step is to decide where 'traffic lights' should be placed to enforce the desired discipline. It is easily found that the following set of lights is minimally required:

- * a light at both ends of each street track, to control the number of trams that will enter the street (at points 1, 2, 3, and 4 in figure 1), and
- * a light at the start of each bridge, to control the number of trams that will enter the finite bridge tracks (at points 2 and 3 in figure 1).

The next step is then to assign state variables to the quantities we want to control in this system. Using the system description of figure 1 we find the following set of six state variables:

T12, for the number of eastbound trams between points 1 and 2,
 T21, for the number of westbound trams between points 1 and 2,
 T23, for the number of eastbound trams between points 2 and 3 (on the bridge),
 T32, for the number of westbound trams between points 2 and 3 (on the bridge),
 T34, for the number of eastbound trams between points 3 and 4,
 and finally
 T43, for the number of westbound trams between points 3 and 4.

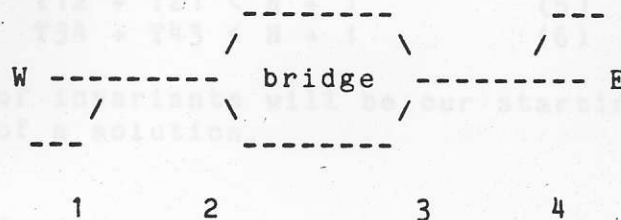


figure 1

We will structure the monitor procedures to be derived into the following three parts:

- (a) a sluice, which holds the wait condition and the queueing code,
- (b) an effect clause, which specifies the effect of the execution of the monitor procedure expressed in state variables, and
- (c) a signaling part, which arranges for the dequeuing of waiting trams that can be released.

To be able to update the state variables we have defined we can expect to need at least four monitor procedures: one for every point in the system (figure 1). The effect clauses in these four monitor procedures will present little difficulty, so we will not pay too much attention to these in the first phase of the design.

The sluices are much more difficult, and we will continue our efforts to arrive at a solution with a discussion of the requirements for the wait conditions in these sluices.

2. Phase I: The Derivation of a Coordination Scheme

2.1 First Attempt

2.1.1 State Invariants

The very least we have to do in our control algorithm is to prevent collisions in the street, and to prevent the tracks from overflowing. These requirements can be expressed as invariants:

$$\begin{aligned}T_{12} * T_{21} &= 0 & (1) \\T_{34} * T_{43} &= 0 & (2) \\T_{23} &< M + 1 & (3) \\T_{32} &< M + 1 & (4) \\T_{12} + T_{21} &< N + 1 & (5) \\T_{34} + T_{43} &< N + 1 & (6)\end{aligned}$$

This set of invariants will be our starting point for the design of a solution.

2.1.2 Wait Conditions

Table I lists the potential effect on the invariants of passing each of the four points in figure 1. An entry 'W' in row p, column (i) means that a tram passing point p from E to W may cause a violation of invariant (i). Similarly, an entry 'E' refers to a tram driving from W to E.

pnt\inv	(1)	(2)	(3)	(4)	(5)	(6)
1	E				E	
2	W		E		W	
3		E		W		E
4		W				W

table I

Using table I we can make a list of conditions under which the violations will actually occur:

$$\begin{array}{ll}1,(1): & T_{21} > 0 \\2,(3): & T_{23} = M \\3,(4): & T_{32} = M \\1,(5): & T_{12} = N \\3,(6): & T_{34} = N\end{array}\quad\begin{array}{ll}2,(1): & T_{12} > 0 \\3,(2): & T_{43} > 0 \\4,(2): & T_{34} > 0 \\2,(5): & T_{21} = N \\4,(6): & T_{43} = N\end{array}$$

list 1

The last two lines in list 1 can be written in a more general form, but as violations of invariant (1) are listed separately, we may as well ignore them here. For a still more complicated problem, we can also first make a list of the precise effects of passing each point in the system; and then correlate this

information with each invariant to obtain list 1.

Deriving wait conditions for the trams from list 1 is straightforward.

In list 2 the symbol 1E refers to the wait condition for a tram arriving at point 1, driving from W to E; the other symbols are defined similarly.

1E: $T_{21} = 0$ and $T_{12} < N$

2E: $T_{23} < M$

3E: $T_{43} = 0$ and $T_{34} < N$

4W: $T_{34} = 0$ and $T_{43} < N$

3W: $T_{32} < M$

2W: $T_{12} = 0$ and $T_{21} < N$

list 2

Non-listed combinations (4E and 1W) have a wait condition 'true', implying that the trams need not be delayed in these cases.

2.1.3 Side Effects

To check for unintentional side effects of the coordination scheme derived so far, we can verify that no steady system state can be reached in which all wait conditions are false at the same time (i.e. a state in which no trams are in transit).

Let us first check to see whether there is such a state at all. The potential deadlock is represented by the following composite condition:

$(T_{21} > 0 \text{ or } T_{12} = N) \text{ and } (T_{21} = N \text{ or } T_{12} > 0) \text{ and}$
 $(T_{43} > 0 \text{ or } T_{34} = N) \text{ and } (T_{43} = N \text{ or } T_{34} > 0) \text{ and}$
 $(T_{23} = M) \text{ and } (T_{32} = M)$

States with $T_{21} > 0$ or $T_{34} > 0$ cannot be called 'steady system states', clearly because the street track can always be emptied of outgoing traffic (the wait conditions 4E and 1W are always true). Therefore, the above composite reduces to:

$(T_{12} = N \text{ and } T_{43} = N) \text{ and } (T_{23} = M \text{ and } T_{32} = M)$

Unfortunately, this state not only exists, but is also very reachable. It is not difficult to find execution sequences that lead into the deadlock. Note that the system, with the given coordination scheme, is bound to deadlock when the following state is reached:

$(T_{12} + T_{23} > M) \text{ and } (T_{43} + T_{32} > M)$

2.2 Second Attempt: avoiding deadlocks

2.2.1 New Invariants

We can avoid the deadlock discovered in section 2.3 by including a seventh system invariant in our set of constraints:

$$(T_{12} + T_{23} < M + 1) \text{ or } (T_{43} + T_{32} < M + 1) \quad (7)$$

2.2.2 Wait Conditions

From the new set of constraints we derive the following set of wait conditions:

$$\begin{aligned} 1E: & (T_{12} + T_{23} < M \text{ or } T_{43} + T_{32} < M + 1) \\ & \text{and } T_{21} = 0 \text{ and } T_{12} < N \end{aligned}$$

$$2E: T_{23} < M$$

$$3E: T_{43} = 0 \text{ and } T_{34} < N$$

$$\begin{aligned} 4W: & (T_{43} + T_{32} < M \text{ or } T_{12} + T_{23} < M + 1) \\ & \text{and } T_{34} = 0 \text{ and } T_{43} < N \end{aligned}$$

$$3W: T_{32} < M$$

$$2W: T_{12} = 0 \text{ and } T_{21} < N$$

list 3

2.2.3 Side Effects

We can now repeat the analysis of section 2.3, to check for unintended side effects of the new coordination scheme. The deadlock state, of course, still exists, but hopefully we will be able to show that it is unreachable now.

The proof can be done in two steps. First we show that the deadlock state cannot be reached if state

$$(T_{12} + T_{23} = M + 1) \text{ and } (T_{43} + T_{32} = M + 1) \quad (a)$$

cannot be reached. Then we show that state (a) is unreachable.

Since the deadlock state can be written as:

$$(T_{12} + T_{23} = M + N) \text{ and } (T_{43} + T_{32} = M + N) \quad (b)$$

the first part of the proof is rather trivial. However, to show that state (a) is itself unreachable we must convince ourselves that there is no execution sequence that leads from initial state

$$(T_{12} + T_{23} = 0) \text{ and } (T_{43} + T_{32} = 0) \quad (c)$$

to (a), if the wait conditions are observed.

Note that for (a) to be reached when starting in (c) there must have been an intermediate state

$$((T_{12} + T_{23} = M) \text{ and } (T_{43} + T_{32} = M + 1)) \quad (d)$$

or

$$(T_{12} + T_{23} = M + 1) \text{ and } (T_{43} + T_{32} = M)) \quad (e)$$

In case (d) the transition to (a) would have been blocked by wait condition 1E; in case (e) the transition would have been blocked by 4W (both conditions from list 3) Q.E.D.

Starvation

The new scheme cannot deadlock, but there may still be other unintended side effects of the coordination.

For instance, we may check that none of the wait conditions can remain false permanently, to secure the requirement of finite delays. For this we will have to consider each wait condition in turn, assume it can be false permanently, and attempt to derive a contradiction from this assumption. As the system is symmetrical, though, we can limit our analysis to just three of the wait conditions: 1E, 2E, and 3E.

2E

Let's start with a simple case. Assume that the wait condition 2E is permanently false:

$$T_{23} = M$$

The blocking is only effective if there are indeed trams waiting at 2E, so we must have:

$$T_{12} + T_{23} > M$$

If the wait conditions are observed this implies that:

$$T_{43} + T_{32} < M + 1$$

Therefore, wait condition 4W must be false. If $T_{43} > 0$ wait condition 3W will remain true until all trams counted in T_{43} have reached the bridge. (We know they will because trams are assumed to make finite non-zero progress.)

No trams can enter the street via 4W, and all trams counted in T_{43} will leave track 43 within finite time. Within finite time, then, T_{43} will be zero.

This means that, also within finite time, wait condition 3E must become true, allowing a tram counted in T_{23} to leave the bridge, causing T_{23} to be decremented, and the trams at 2E to be unblocked. Wait condition 2E can therefore not remain false permanently, Q.E.D.

1E and 3E

Unfortunately, a similar reasoning for 1E and 3E doesn't work. We quickly find that both 1E and 3E can in fact remain false forever if the trams in the opposite direction succeed in keeping an overlapping block on the incoming and outgoing street tracks. Nothing in the present coordination scheme can prevent this from happening.

A starvation of this type is of course unlikely to occur in the real Leidsestraat for any length of time, as tram drivers will quickly recognize the problem and avoid it. Still, also a temporary livelock of trams can cause longer-lasting disturbances in the time schedules due to domino effects (clustering of trams). It may therefore be worth our while to try and avoid also starvation in our design.

2.3 Third Attempt: avoiding starvation

Starvation is a dynamic problem. It cannot be solved elegantly by introducing new system invariants. (System invariants are meant to formalize static coordination requirements. For a discussion of this point see HOLZ79 or HOLZ80.)

We can try to avoid starvation by using a priority flag for each street track. If trams are waiting at both ends of a track the priority flag will decide which side is allowed to enter the track, and which is to wait. Starvation, of course, is only avoided if we also make sure that the priorities are swapped regularly. We could swap the priority after a certain fixed number of trams has used the flag to resolve a tie. We could also swap priorities each time a tram reaches the end of a track. The priority is then always to the disadvantage of the direction that used the track most recently.

We adopt the second method here, by virtue of its simplicity. Let's name the priority flag for the west track (between point 1 and 2) PW, and name the priority flag for the east track PE. Each flag can have one of two values: E or W. Assuming that initialization and swapping of priorities can be taken care of later, we can readily derive the new wait conditions.

We do need some way of measuring the length of a waiting queue. In our target language Concurrent Pascal we have such a tool, so we do not have to worry about implementation problems here. We will use the symbol $Q(pD)$ to indicate the number of trams with direction D delayed at point p.

We add a few clauses to the conditions in list 3:

to 1E: $Q(2W) > 0$ or $PE = E$
to 2W: $Q(1E) > 0$ or $PE = W$
to 3E: $Q(4W) > 0$ or $PW = E$
to 4W: $Q(3E) > 0$ or $PW = W$

list 4

Since the additions have only tightened the coordination scheme a bit more, we may expect that states that were unreachable before are still unreachable, most notably the deadlock state. Note also that the new clauses cannot all be true at the same time, and thus cannot introduce new systemwide deadlocks as such.

The argument to show that the new scheme is actually deadlock- and starvation-free can be repeated now. For brevity we skip it here.

To complete the analysis we may also want to convince ourselves that the scheme is 'minimal'. We can try to show that any violation of a wait condition implies a violation of either a system invariant (formalizing static requirements) or a design rule (formalizing dynamic requirements). Of course, in doing so we merely push the verification problem to another level: it then remains to be shown that also the system invariants and the design rules are minimal. We will avoid a lengthy discussion of this topic by frivolously claiming the result to be trivial.

3. Phase II: The Design of a Signalling Scheme

3.1 Requirements and Assumptions

We have completed the first phase of the design: we have derived a minimal and consistent set of coordination rules, expressed in wait conditions. In the next phase we will have to complete the design with a minimal and complete signaling scheme: a methodology for queueing and dequeuing trams.

The requirements for the signaling scheme depend on the target language for the coordination. For Concurrent Pascal the requirements are as follows:

- * Once a process is delayed it will not resume execution automatically: it will have to be signaled explicitly.

- * Only one process can be signaled at a time. Issuing a wake-up signal implies that the signaling process loses the privilege to execute the monitor code (via which the coordination is enforced) in favor of the process signaled.

We further adopt the rule that:

- * A process will not reevaluate its wait condition when it is signaled: it will assume the wait condition to be true, and continue executing. The signaling process will therefore have to make sure that the wait condition of the process to be signaled is true.

To design a signaling scheme we can choose from three different approaches:

- (a) We make a list of all the queues in the coordinated system (in our case 6). Each passage (i.e. each tram passing a point where system state variables are affected) will cause a re-valuation of the wait conditions for the trams delayed in these queues, in a predetermined fixed order. If no tram (process) from the first queue can be signaled, either because the corresponding wait condition is still false, or because the queue is empty, the next queue in the priority list is evaluated. If necessary all queues are considered, but as soon as a wake-up signal has been issued the evaluation process is aborted.

In the tram system the priorities can be chosen as follows:

- * first try to clear the street tracks by allowing trams to enter the bridge (the queues at 2E and 3W),

* then try to empty the bridge tracks (3E and 2W),
 * and only if neither of these signals can be
 issued allow new trams to enter the street (1E and
 4W).

(b) Each passage only causes the re-evaluation of
 wait conditions that may have been made true by
 the corresponding effect on the state variables.
 If there are more candidates for signaling the
 queue associated with the passage itself will have
 the highest priority: the other candidates are
 considered in a fixed order.

(c) As (b), but here the evaluation priority of
 the queue associated with the passage itself is
 lower than the priority of all other candidates.
 Pending signaling tasks are transferred from
 signaling process to signaled process (see below).

The third approach opens up a whole new area of
 interesting problems. Enough reason for a closer look.

3.2 The Transfer of Signaling Tasks

In order not to complicate matters right from the
 start, let us first pretend that we do have a way of
 signaling more than one delayed tram (process) at a
 time, but only if they are delayed in the same queue
 (i.e. at the same point in the same direction). Let
 us also pretend that each tram will re-evaluate its
 wait condition before proceeding. Thus, we can first
 make sure where the signals should be issued, and defer
 the problems related to the question of how they should
 be issued within the constraints of Concurrent Pascal.

A useful starting point is table II, which specifies
 which passages can cause delayed trams to be released.
 A tram passing point 1 driving from E to W, for
 instance, (1W in the table) will decrement T21, and may
 thereby remove a block on trams delayed at 1E.

pass\cond	1E	2E	3E	4W	3W	2W
2E	+					+
3E		+				
4E				+		
3W			+	+		
2W					+	
1W	+					

table II

Not listed passages (1E and 4W) cannot remove blocks;
 not listed conditions are true (1W and 4E).

(Note: consider why combinations like 3E,4W and 3E,1E
 are not included in the table !)

If passing a point does remove a block for a waiting tram that tram will have to be signaled. In table 2 we see that in two cases there is more than one queue candidate for signaling (passing 2E or 3W). Still, only one signal to a single queue can be issued.

Fortunately, in these cases there is no conflict between the candidates for signaling in either case. Consider 2E. If $Q(2W) > 0$ queue 1E need not be signaled at all, because queue 2W will take priority. If $Q(2W) = 0$, clearly queue 2W need not be signaled as there are no trams delayed there.

We thus arrive at the following signaling scheme (figure 2).

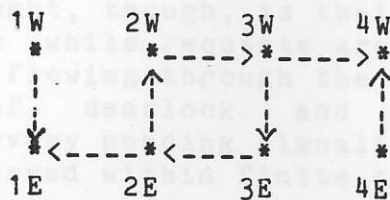


figure 2

3.3 Replacing Queue Signaling with Process Signalling

Of course, we do not have queue signaling in Concurrent Pascal, so we will have to refine our signaling scheme. We have quite a problem at this point since we should still be able to trigger the release of more than one process (from the same queue) by issuing just a single signal.

We want to find out how we can transfer signaling tasks from one process to the next: from signaling process to signaled process. There is, however, a danger of triggering an avalanche of signal transfers, which cannot all be served in proper order.

To get a feeling for the nature of the problems encountered here, let us consider the signal from a tram passing point 2E to delayed trams at 2W.

If the tram passing 2E clears the west track of the street, one or more trams delayed at 2W can enter that track in the opposite direction. The trams in queue 2W can enter the track at least until the first of them reaches point 1, and reverses the priority flag PW.

Still, we can only signal one tram in queue 2W. The first tram in line will receive the signal, together with the (transferred) task to signal the others. When the signaled tram passes 2W, though, it is expected to signal not only its successor in line at 2W (as implied

by the transferred signaling task), but also the first tram delayed at 3W. (Note that a tram leaving the bridge track makes room for precisely one other.) So we have a conflict.

We can transfer the pending signaling task again, from 2W to 3W, but trams passing 3W already have the task of either signaling delayed trams at 4W or trams delayed at 3E. If 4W is signaled the problem is solved, as 4W has no other obligations to signal queues. But, if 3E is signaled we're in trouble because the signaling task may then migrate back to 2E, where it originated. (To aggravate the problem, signaling 3E may trigger another series of pending signaling tasks.) The danger is that the pending tasks move around endlessly, but are never served.

A comforting thought, though, is that the pending tasks can only circle while requests are being served, and while trams keep flowing through the system. Thus, in the absence of deadlock and starvation every tramprocess and every pending signaling task will in fact always be served within finite time.

The treatment of pending signaling tasks can be expected to notably influence system throughput, so that it would surely be worthwhile to develop a design discipline that would guide us to an optimal scheme. For lack of such a design method we will content ourselves with a signaling scheme that is both 'minimal' and 'complete', but not necessarily 'optimal' (HOLZ80).

A signaling scheme is called minimal if no process is signaled that cannot proceed.

A signaling scheme is called complete if any process that can continue will be signaled within finite time.

To realize such a design we extend table II with all transferred signals. We then assign priorities wherever there is a conflict between two or more signaling tasks. The result of this effort is depicted in table III, where pluses are replaced by numbers (low numbers correspond to high priorities).

To simplify the signaling scheme somewhat we ignore the finiteness of the street tracks from this point on (bound N).

pass\cond	1E	2E	3E	4E	4W	3W	2W	1W
1E	1*							
2E			2*				1	
3E		1	3*				2*	
4E					1			
4W					1*			
3W			1				2*	
2W			2*			1	3*	
1W	1							

table III

The stars in table III label the transferred signaling tasks. The signaling scheme that corresponds to table III is given below in figure 3.

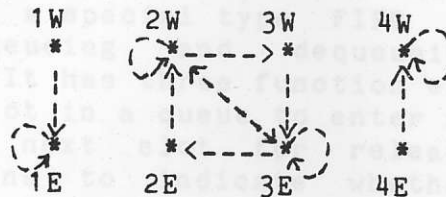


figure 3

(Note that signals relating to the finiteness of the street tracks have been deleted: signals from 2E to 1E, and from 3W to 4W).

4.1 Program Listing

```

*****
* N is size of bridge tracks, T is number
* of trains per direction.
*****

```

```

CONST N = 3; T = 30;

```

```

*****
* Type FIFO will take care of queuing and *

```

4. A Solution in Concurrent Pascal

This section contains a listing of (the more interesting parts of) a program text derived with the design method discussed in this report. The text was written to illustrate the point that a working solution can be derived in a systematic manner. Shortcuts for efficiency are therefore willfully ignored.

The wait conditions from the design translate into queueing conditions in Concurrent Pascal. The complements of the conditions from lists 3 and 4 can be found in the program text (note that the capacity of the street tracks is now taken infinite). The structure of the text is as follows.

After the declaration of two system constants, one for the size of the bridges and one for the number of trams to be generated, a special type FIFO is declared to order the queueing and dequeuing of delayed tram processes. It has three function entries: one to find the next slot in a queue to enter a process (INN), one to find the next slot for releasing a process (OUT), and one to indicate whether or not the associated queue is empty (WAITING).

The monitor itself is declared as a special data type named L. It has a number of variables to count the number of trams (the state variables), and four procedures to guard the passages of the four points in the system (fig. 1). Note that each procedure entry has three distinct parts: a waiting sluice, an effect clause, and a signaling part.

The wait conditions themselves are specified as function entries.

Finally, we declare a tram as a special data type 'process'.

The actual program body is relatively simple: it merely initializes the instances of all the data types that have been declared.

4.1 Program Listing

```
*****
* M is size of bridge tracks, T is number *
* of trams per direction.                  *
*****
```

```
CONST M = 3; T = 30;
```

```
*****
* Type FIFO will take care of queueing and *
```

* dequeuing of trams. *

TYPE FIFO = CLASS (LIMIT: INTEGER);
VAR HEAD, TAIL, SIZE: INTEGER;

FUNCTION ENTRY INN: INTEGER;
BEGIN INN := HEAD;
 HEAD := HEAD MOD LIMIT + 1;
 SIZE := SIZE + 1
END;

FUNCTION ENTRY OUT: INTEGER;
BEGIN OUT := TAIL;
 IF SIZE > 0 THEN
 BEGIN TAIL := TAIL MOD LIMIT + 1;
 SIZE := SIZE - 1
 END
 END
END;

FUNCTION ENTRY WAITING : BOOLEAN;
BEGIN WAITING := (SIZE > 0) END;

BEGIN HEAD := 1; TAIL := 1; SIZE := 0 END;

* Direction is either east or west *

TYPE GOAL = (EAST, WEST);

* Declaration of the monitor procedures. *
* The direction of a tram will be used to *
* index arrays. INSC counts trams between *
* points 1 and 2, ONBC between 2 and 3, *
* and OUTC between 3 and 4. There are three *
* queues per direction, and thus we need *
* three instances of class FIFO per direct- *
* ion. Array PRIOR holds the two priority *
* flags that avoid starvations. *

TYPE L = MONITOR
VAR INSC, OUTC: ARRAY [GOAL] OF INTEGER;
 ONBC: ARRAY [GOAL] OF 0 .. M;
 INSQ, ONBQ: ARRAY [GOAL, 1..T] OF QUEUE;
 OUTQ: ARRAY [GOAL, 1..M] OF QUEUE;
 INSF, OUTF, ONBF: ARRAY [GOAL] OF FIFO;
 PRIOR: ARRAY [GOAL] OF GOAL;
 I: GOAL;

FUNCTION OPP (DIR: GOAL): GOAL;
BEGIN IF DIR = EAST THEN OPP := WEST ELSE OPP := EAST END;

```

FUNCTION WIN (DIR: GOAL): BOOLEAN;
BEGIN WIN :=
  (OUTC [OPP(DIR)] > 0) OR
  (( INSC [OPP(DIR)] + ONBC [OPP(DIR)] > M) &
   ( INSC [DIR] + ONBC [DIR] = M)) OR
  (OUTF [OPP(DIR)].WAITING & (PRIOR [DIR] = OPP(DIR)))
END;

FUNCTION WON (DIR: GOAL): BOOLEAN;
BEGIN WON := (ONBC [DIR] >= M) END;

FUNCTION WOF (DIR: GOAL): BOOLEAN;
BEGIN WOF :=
  (INSC [OPP(DIR)] > 0) OR
  ((INSC [OPP(DIR)].WAITING & PRIOR [OPP(DIR)] = OPP(DIR))) &
  "*** check for deadlock on opposite side !! ***"
  (INSC [DIR] + ONBC [DIR] <= M) OR
  (INSC [OPP(DIR)] + ONBC [OPP(DIR)] < M))
END;

*****
* Start of declaration monitor procedures *
*****

PROCEDURE ENTRY INS (DIR: GOAL);
BEGIN
  IF WIN(DIR) THEN DELAY (INSQ [DIR, INSF [DIR].INN]);

  INSC [DIR] := INSC [DIR] + 1;

  IF (INSF [DIR].WAITING & NOT WIN(DIR))
  THEN CONTINUE(INSQ [DIR, INSF [DIR].OUT])
END;

PROCEDURE ENTRY ONB (DIR: GOAL);
BEGIN
  IF WON(DIR) THEN DELAY (ONBQ [DIR, ONBF [DIR].INN]);

  INSC [DIR] := INSC [DIR] - 1;
  ONBC [DIR] := ONBC [DIR] + 1;
  PRIOR [DIR] := OPP (DIR);

  IF ((INSC [DIR] = 0) & (OUTF [OPP(DIR)].WAITING))
  THEN CONTINUE (OUTQ [OPP(DIR), OUTF [OPP(DIR)].OUT])
  ELSE
    IF (OUTF [DIR].WAITING & NOT WOF(DIR))
    THEN CONTINUE (OUTQ [DIR, OUTF [DIR].OUT])
  END;

PROCEDURE ENTRY OFF (DIR: GOAL);
BEGIN
  IF WOF(DIR) THEN DELAY (OUTQ [DIR, OUTF[DIR].INN]);

  ONBC [DIR] := ONBC [DIR] - 1;

```

```

    OUTC [DIR] := OUTC [DIR] + 1;

    IF ONBF [DIR].WAITING
    THEN CONTINUE (ONBQ [DIR, ONBF [DIR].OUT])
    ELSE
        IF (OUTF [OPP(DIR)].WAITING & NOT WOF(OPP(DIR)))
        THEN CONTINUE (OUTQ [OPP(DIR), OUTF [OPP(DIR)].OUT])
        ELSE BEGIN IF OUTF [DIR].WAITING THEN
            CONTINUE (OUTQ [DIR, OUTF [DIR].OUT])
        END
    END;

PROCEDURE ENTRY OUTS (DIR: GOAL);
BEGIN
    OUTC [DIR] := OUTC [DIR] - 1;
    PRIOR [OPP(DIR)] := OPP(DIR);
    TOTAL := TOTAL + 1;

    "*** check for deadlock condition !! ***"
    IF ((INSC [DIR] + ONBC [DIR] > M) &
        (INSC [OPP(DIR)] + ONBC [OPP(DIR)] = M))
    THEN
        BEGIN
            IF (OUTF [DIR].WAITING)
            THEN CONTINUE (OUTQ [DIR, OUTF [DIR].OUT])
        END
    ELSE IF ((OUTC [DIR] = 0) & INSF [OPP(DIR)].WAITING)
    THEN CONTINUE (INSQ [OPP(DIR), INSF[OPP(DIR)].OUT])
END;

*****
* Initialization of the monitor *
*****

BEGIN
    FOR I := EAST TO WEST DO
    BEGIN
        INSC[I] := 0; OUTC[I] := 0; ONBC[I] := 0;
        INIT INSF[I](T), OUTF[I](M), ONBF[I](T);
        PRIOR[I] := I;
        PENDING[I] := FALSE
    END
END;

*****
* Another data type is the tram-process. *
*****

TYPE TRAMS = PROCESS (DIR: GOAL; STREET: L);
BEGIN
    REPEAT WITH STREET DO
        BEGIN INS(DIR); ONB(DIR); OFF(DIR); OUTS(DIR) END;
    UNTIL (FALSE);          "**** FOREVER ****"

```

END;

```
*****  
* Start of the main program body *  
*****
```

```
VAR LSTREET: L;           "an instance of the monitor"  
    TRAM   : ARRAY [1 .. T] OF TRAMS;  "T tram processes"  
    J : 1 .. T;  
  
BEGIN                               "initializations"  
    INIT LSTREET; J := 1;  
    REPEAT INIT TRAM [J](EAST, LSTREET);  
           INIT TRAM [J+1](WEST, LSTREET);  
           J := J + 2;  
    UNTIL (J > T);  
END.
```

To convince ourselves that the signaling scheme developed is 'minimal' we can check that for every process that receives a signal the corresponding wait condition is false.

An argument for the 'completeness' of the signaling can be given by assuming a delay of trams in either one of the system queues, and claiming the delay to be permanent. A contradiction should then be derived from that assumption.

These proofs are quite similar to the ones presented earlier in the paper, and are omitted here.

I presented the problem to the students in a class on Operating Systems theory, as an optional assignment for their term-papers. Many students took the challenge, and in the weeks that followed we discovered the complexities of the problem, and (more importantly) the importance of systematic design.

I gratefully acknowledge the students at USC in Los Angeles who participated in class CS 555, in the spring of 1980.

I thank Susan Masahito for forcing me to write 'pluses' instead of 'plusses', and I apologize to her for not writing "pluses".

5. Conclusion

Coordination schemes for systems of interacting concurrent programs can be designed via a systematic design discipline. Solutions and intermediate attempts to arrive at solutions can be (dis)proved convincingly within this discipline.

Though the example in this report is directed at the design of a solution in monitor structures for a Concurrent Pascal program, the design discipline is by no means restricted to any particular target language or synchronizing primitive.

The design in this report illustrates how coordination schemes can be designed systematically even without strictly adhering to the formalisms developed in HOLZ80.

Acknowledgement

My original reason for attempting to model the Leidsestraat problem in a Concurrent Pascal solution was to provide a simple example for a talk on concurrent program design. Too late I found that the example was much better than I had intended it to be. Unfortunately, this meant that, though a convincing solution could be described on a few viewgraphs, a correct solution was much harder to find. I presented the problem to the students in a class on Operating Systems theory, as an optional assignment for their term-papers. Many students took the challenge, and in the weeks that followed we discovered the complexities of the problem, and (more importantly) the importance of systematic design.

I gratefully acknowledge the students at USC in Los Angeles who participated in class CS 555, in the spring of 1980.

I thank Susan Massotty for forcing me to write 'pluses' instead of 'plusses', and I apologize to her for not writing "pluses".

6. References

HOLZ79, 'Coordination problems in multiprocessing systems.' Ph.D. Thesis, Delft University of Technology, June 1979, 315 pp.

HOLZ80, 'The design of coordination schemes.' Computing Science Technical Report No. 87, Bell Laboratories, Murray Hill, NJ, August 1980, 25 pp.