# Algebraic specification of a system for traffic regulation at signalized intersections

S.F.M. van Vlijmen
A. van Waveren
*University of Amsterdam, Programming Research Group,*
*Kruislaan 403, 1098 SJ Amsterdam, The Netherlands.*
*E-mail:* vlijmen@fwi.uva.nl — waveren@fwi.uva.nl

**Abstract**

A specification of a part of an existing system for traffic regulation at signalized intersections is given. The system that is chosen as subject to specification is developed by the dutch company Nederland Haarlem. The specification, in PSF, centers on the algorithm performing the regulation, which is, in our view, the most interesting part. The specification given is data-oriented, i.e., there is not much emphasis on processes. Although time plays an important role in the regulation process, this is not an obstacle for specification.

1991 Mathematics subject classification: 68Q40, 68Q45
1990 CR categories: D.1.3, D.2.1, F.1.2.
Key words & phrases: algebraic specification, process algebra, PSF, traffic regulation.

## 1 Introduction

Formal methods are a research topic in the theoretical computer science for some twenty-five years by now. One of the applications of formal methods is found in the field of formal specification. Central notion here is the formal specification language, its syntax and its semantics. A formal specification is a valid text stated in a formal specification language. To our opinion, research in the field of formal specification has led to formal specification languages that are practically applicable; there are a number of quite expressive formal specification languages built on reasonable stable theories and which have appropriate tool support. In order to show the applicability of formal specification languages, we try to apply them in various situations. As part of this activity this report is written.

Although the research on formal specification is an international enterprise we will now shortly concentrate on some results of the University of Amsterdam and the Centre for Mathematics and Computer Science, also in Amsterdam.

At the Programming Research Group of the University of Amsterdam and the research groups AP2 and AP3 of the Centre for Mathematics and Computer Science research is and has been conducted on process theories and algebraic specification. Over the years this has led to a number of formal specification languages with appropriate tool support, to name some: ASF, PSF and μCRL [BHK89], [M91], [GP90, GP91]. For PSF this amounts to a syntax and type checker, a termrewriter, a simulator and a proof assistant. The availability of PSF and the tools for PSF make it possible to use the theories on which the language is based on a larger scale and in a more practical way than before in industry. These contacts lead to the necessary feedback, new research issues and funding for further research. To give some examples, a number of communication protocols have been investigated [MM90], [B91], [W91]; work has been done on CIM (computer instructed manufacturing) [M91], [HP92], [PV93]; in the field of telecommunication networks [KVW93] and banking [SS93]. This report is about the specification of an existing system: a traffic regulation system for signalized intersections built by the dutch company Nederland Haarlem[1] [K90].

Nederland Haarlem started with the manufacturing of miner's lamps, about seventy years ago. Today Nederland Haarlem has about fivehundred employees and is active in a number of markets: control systems for the petrochemical industry, street furniture (e.g. traffic signs), and say 'traffic management systems'. Currently Nederland Haarlem is market-leader in the Netherlands in the field of these traffic management systems. These systems comprise systems for the flow control on highways, systems for congestion detection, systems for regulation of traffic at intersections etc.. It is one of their systems for traffic regulation at signalized intersections we took as subject for specifications in PSF (Process Specification Formalism [MV89, MV90, M91]).

In this text we give only a short and informal introduction to PSF. For information on the theories underlaying PSF we refer to [BW90, BHK89, M91].

In section 2 traffic regulation systems for signalized intersections are introduced. In section 3 an example is given, in order to illustrate the approach to regulation of traffic as introduced in the section 2. Section 4 contains the informal introduction to PSF. The part of the system we selected to specify and the assumptions we made are given in section 5. In section 6, a discussion on the most important data and process modules is given. We finish with some comments and concluding remarks in section 7.

---

[1]Nederland Haarlem, P.O.Box 665, 2003 RR Haarlem, the Netherlands.

Extra examples in the style of the example in section 3 are given in appendix A. The listing of the specification is given in appendix B and C. This listing is incomplete in the sense that modules imported from the PSF-library by Jos van Wamel are not listed [W93].

## Acknowledgements

Herewith we wish to thank Jan Kroone and Henk Spronk of Nederland Haarlem, for the cooperation. Jan and Henk frequently took time to answer our questions, and they supplied us with clear and concise documentation.

Furthermore we wish to thank Jan Bergstra, Ard Overkamp, Jan van Schuppen and Jos van Wamel for various contributions and Sjouke Mauw for helpful comments on the specification and on PSF.

## Contents

**1 Introduction**

**2 Systems for traffic regulation at signalized intersections**

2.1 General

2.2 The four order regulation system

2.3 Regulation algorithm of the four order regulation system

**3 Example of four order regulation**

**4 Introduction to PSF**

4.1 Data modules and process modules

4.2 Parameter binding

**5 The specification: choices, assumptions and solutions**

5.1 Components selected for specification

5.2 The data set

5.3 The control process

5.3.1 The detection devices

5.3.2 Priority

5.3.3 Green duration

**6 The main modules of the specification**

6.1 Module `Phase` and module `Phase-Set`

6.2 Taking account of detection, priority and time

6.2.1 Module `Account-For-Detection-Operation`

6.2.2 Module `Account-Weighed-Priorities-Operation`

6.2.3 Module `Account-For-Time-Operation`

6.3 First, second and third order realization
6.3.1 Module `Realization-Toolkit`
6.3.2 Module `Realization-First-Order`
6.3.3 Module `Realization-Second-Order`
6.3.4 Module `Realization-Third-Order`
6.4 Module `Control-Process`
6.5 Module `Traffic-Regulation-System`
**7 Comments and future work**
7.1 Comments
7.2 Future work
**References**
**A Example simulations of the specification**
**A.1 Example 1**
**A.2 Example 2**
**A.3 Example 3**
**A.4 Example 4**
**B The specification**
**C Example of an actual situation data set**

# 2 Systems for traffic regulation at signalized intersections

## 2.1 General

With a signalized intersection we mean an intersection where every stream of traffic participants (stream, for short) on an approach to the intersection is regulated with a *signal*.

Traffic regulation systems for signalized intersections can be categorized on their dynamic behaviour.

At the one end, one has rigid regulation systems that have no dynamic behaviour at all. In these systems all durations for green, yellow and red of the signals are fixed. And the order in which the signals are turned is fixed too. Every step in the scheduling is explicitly stated.

At the other end of the spectrum we have regulation systems that determine dynamically, on basis of incoming requests, what action to take next. Here is explicitly stated what one does not want to see actuated: e.g. conflicting streams of participants. The four order system, developed by Nederland Haarlem, falls in this last category. We have to stress here that four

order regulation is a Nederland Haarlem proprietary notion. We did not come across an accepted naming for categories of regulation systems in the literature. It is the four order regulation system that is the subject of this study.

## 2.2 The four order regulation system

The four order regulation system is in this subsection introduced in its general form. See section 5 for the part that is actually specified. In this section, words in *italic* refer to elements in figure 1.

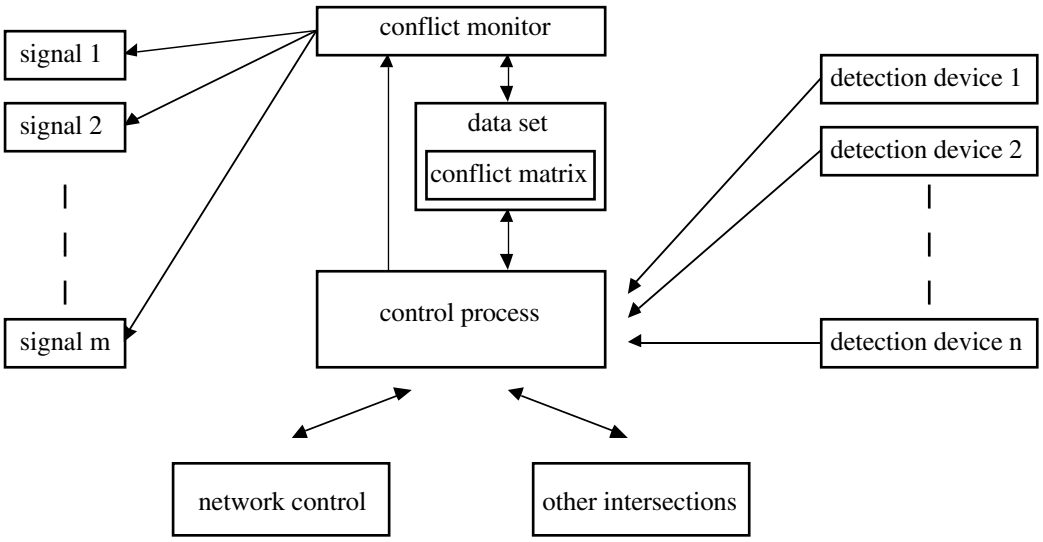


figure 1

The four order regulation system is a generic system. The *data set* turns the generic regulation system into a regulation system for a specific intersection.

The system has *detection devices* that have at least the ability to detect traffic participants at the approaches. Optional detection devices can moreover detect speed or type of traffic participants. The type of the traffic participant is, for instance, meaningful if one wants to be able to detect an ambulance. The system regulates, however, not streams but phases, i.e., sets of streams receiving identical signals from the *control process*.

Given an intersection with a set of streams S, the set of phases is a partition of S, where the streams in a phase are not conflicting, for the sake of safety. It is important to note that the set of phases is *fixed*; the partition of S does not change dynamically. Maybe the word phase, typical traffic vernacular, is a bit confusing. The word phase should not be interpreted as some state the intersection is in at a certain point in time. Phases are the objects that are regulated, their composition, i.e., the (sub)set of the streams they consists of is fixed in time. In all the examples, however, the phases mainly consist of only one stream. For the control

process it does not matter that much how many streams there are in a certain phase. If, for instance, a phase should be turned green, the control process just outputs a command to instruct some hardware to switch the signals of all the streams in the phase.

Introduction to the four orders is postponed to subsection 2.3. For now, think of an order as some set of circumstances specifying when (the signals of) a phase can be turned green. Conflicts between phases are recorded in the *conflict matrix*. See figure 2b below, for an example. The conflict matrix is part of the data set.

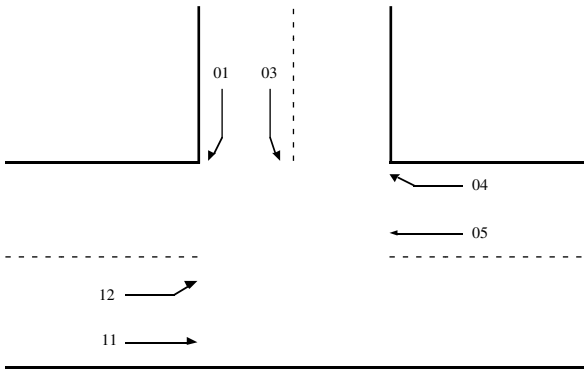


figure 2a

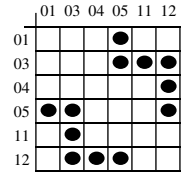


figure 2b

The *conflict monitor* uses the conflict matrix. It filters commands sent by the control process to the signals. In case a conflicting situation is bound to occur it turns all signals to intermittent yellow and does not respond any more to input from the control process. The name ‘conflict matrix’ is slightly misleading. Actually the conflict matrix contains a lot of extra conditions, and is as such a set of conditions, not only a matrix of conflicting phases as in figure 2b. For instance, the conflict matrix in reality contains information on minimal duration of red, green and yellow. The matrix of conflicting phases presents a symmetric and non-reflexive relation: if phase $x$ has a conflict with phase $y$, then $y$ also has a conflict with phase $x$; and a phase is not in conflict with itself. This fact may lead to a compact implementation. In this paper we do not care about that. The matrices should be easily readable from ‘left to right’ and contain as such twice as much information as needed.

The main task of the control process is the scheduling of phases. It does so on the basis of dynamically determined priorities: there is no pre-defined order in which phases are scheduled. However, conflicts and conditions as laid down in the data set prevent certain combinations. Furthermore, it is possible that the control process is influenced by systems regulating other intersections or by *network control*, i.e., remote control imposed on one or more traffic regulation systems. Often this network control is human operated.

## 2.3 Regulation algorithm of the four order regulation system

On an abstract level the control process is nothing more than a loop in which the following actions take place:

— read detection devices;
— calculate new status of phases (turn green, yellow or red);
— change signals according to the new status.

And then the process starts again.

The core of the control process is found in the functions performing the calculation of the new status of the phases: the regulation algorithm. Before going into the details some Nederland Haarlem vernacular may come in handy.

A phase is *realized* when its signals are turned green. A phase is *requesting realization* if its signals are red and they have been red for some minimal duration, the *clearance-duration*, and it has traffic waiting. A phase which has its signals red for a period longer than the clearance duration is said to be *cleared*. In other words, a phase that is cleared and has traffic waiting is requesting realization.

There are four orders of realization: first order, second order, third order and fourth order. An order is a name for a set of circumstances specifying when a phase can be realized. For instance, if for a certain phase, with respect to the rest of the phases at an intersection, the circumstances for a first order realization are fulfilled, the phase gets a first order realization.

Suppose we can dynamically calculate a priority for the cleared phases, such that these phases do not get the same priority and a phase with no traffic waiting gets a lower priority than a phase which has traffic waiting. Now the intuition with the four orders is the following. The first order specifies the realization of phases that are requesting realization and which have a priority higher than their conflicting phases. (One can have more than one first order realization.) This order is in a sense the backbone of the regulation algorithm because the regulation algorithm realizes phases on their dynamically determined priorities. Second order and third order realizations serve to increase the throughput of the intersection by allowing realization of phases that could not get a first order realization. So, second and third order realizations jump the queue. However, second and third order realizations get regulated in such a way that they will not frustrate a phase of higher priority. To explain this, suppose we have three phases, 01, 03 and 05 in figure 3 below. Phase 03 is currently green, phase 01 and phase 05 are requesting realization. Phase 05 has higher priority than phase 01. Because phase 05 has to wait for phase 03, one can give phase 01 a second order realization, provided that phase 01 does not frustrate phase 05 on the long term. That is, as soon phase 03 turns red, phase 01 should be turned red too.

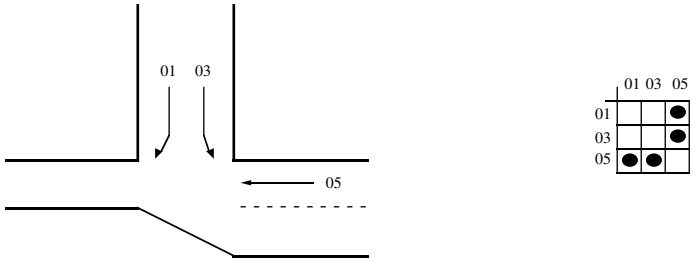


figure 3

What about scheduling fourth order? Fourth order realizations are realizations requested by traffic participants with special priority. For instance realization requests raised by an assistance service (fire patrol, ambulance, police patrol) or a public transport vehicle. But also a request from a regulation system at another intersection falls in this order. And so does a boat requesting a bridge to open! Fourth order realizations get at least a high priority, if not the absolutely highest priority. In that case realization is forced.

As priority is concerned, the control process actually checks if there is a fourth order request for realization *before* handling first order! Probably only from a historical viewpoint the name fourth order can be explained: Nederland Haarlem developed a three order regulation system that got more and more sophisticated. Eventually they had to introduce a new order: why not the fourth order?

In the following, first, second and third order realizations are defined more precisely. This is however still an abstraction. For detail we refer to [K90].

1. A phase will be realized *first order* if:
1.1 it is requesting realization, and
1.2 it has higher priority than any phase also requesting realization with which it is in conflict, and
1.3 it is not in conflict with any of the *not-cleared* phases, i.e., it is not in conflict with a phase which is green or yellow, and it is not is conflict with a phase which is red for a period less than or equal to the clearance duration.

2. A phase will be realized *second order* if:
2.1 it is requesting realization but can not be realized first order, and
2.2 it can be realized without frustrating a phase with a request for realization of higher priority pending, and
2.3 it is not in conflict with any of the not-cleared phases.

3. A phase will be realized *third order* if:
3.1 it is cleared and it has no traffic waiting, and
3.2 it can be realized without frustrating a cleared phase with a higher priority that has no traffic waiting, and
3.3 it can be realized without frustrating a phase with a request for realization pending, and
3.4 it is not in conflict with any of the not-cleared phases.

If a phase *p* is not in conflict with any of the not-cleared phases, then we often say that *p* has no *direct* or no *immediate* conflict. At first glance, there does not seem to be a big difference between second and third order realizations. The important difference is in special conditions in the dataset preventing certain combinations of realizations, these are not considered by us. Furthermore, what we discussed here is only the third realization of type 2. There are other types of third order realization not taken in consideration here, i.e., type 1 and type 3 [K90].

## 3 Example of four order regulation

Suppose we have the twelve phases as in figure 4. All phases are cleared. Phase 08, 06, 11, 01 and 10 have respectively 3, 2, 2, 1 and 1 traffic participant waiting. Suppose we have the following order of decreasing priority: 08, 06, 11, 01, 10, 12, 09, 07, 05, 04, 03, 02. Note that this ordering is not fixed. Furthermore, there are no special traffic participants detected, i.e., there is no need for fourth order scheduling.

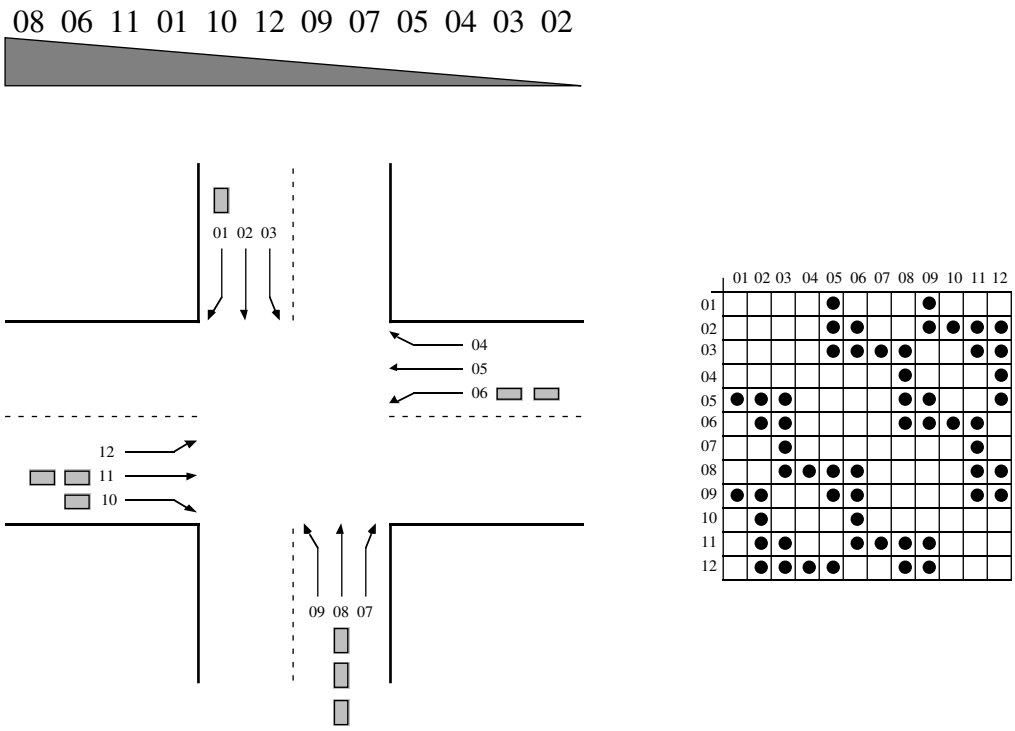


figure 4

The numbers (x.y) in the following refer to the conditions for realization as given in subsection 2.3 above. Phase 08 gets a first order realization: phase 08 is requesting realization (1.1), there is just no phase with higher priority (1.2) and there are no conflicts of the type specified in (1.3) because all other phases are cleared. Subsequently the conflicts of phase 08, i.e., phases in the set {03, 04, 05, 06, 11, 12}, will not be realized now. So, phase 06 is skipped although it is the next in priority, the same holds for phase 11. Phase 01 is now the next in priority, it gets a first order realization because it is requesting realization (1.1), its conflicts, phase 05 and phase 09, are lower in priority and cleared (1.2)(1.3). Phase 10 can not get a first order realization, one of its conflicts, phase 06, is higher in priority. Phase 10 gets a second order realization. We have to make sure, however, that phase 10 does not frustrate 06 on the long term (2.2). If after some time, phase 06 is not blocked by phase 08 any more, then phase 06 should not be blocked by phase 10. This is achieved by making sure phase 10 is cleared before or at the same time as phase 08. The only phase not blocked by now is 07; phase 07 gets a third order realization if we make sure it is cleared before or at the same time as phase 08.

# 4 Introduction to PSF

In this section the specification language PSF is informally introduced by looking at some small specifications. For a more elaborate treatment of PSF we refer to [M91]. This section serves to let the reader get acquainted with PSF before going into the details of the specification in sections 5 and 6.

## 4.1 Data modules and process modules

Suppose we want to regulate in a rigid way the four signals/phases/streams[2] 02, 05, 08 and 11 at the intersection in figure 5 below. The regulation system behaves as follows:
— turn green the signals 02 and 08;
— turn red the signals 02 and 08;
— turn green the signals 05 and 11;
— turn red the signals 05 and 11;
— start all over again.

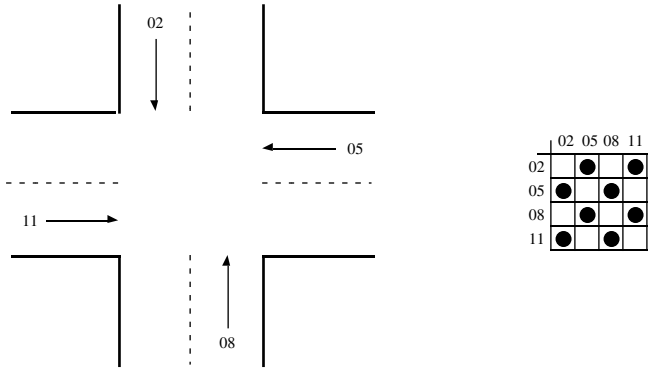


figure 5

The system is specified in two *data modules* (`Signal-Status, Signal-Name`) and one *process module* (`Traffic-Example`). In a data module one specifies *sorts* and *functions* that operate on these sorts. In data module `Signal-Name` sort `SIGNAL-NAME` is declared together with four constants of sort `SIGNAL-NAME`, all in the *export* section. Data module `Signal-Name` is similar. But here is a function `turn` declared which is axiomized in an *equations* section. The axioms are the rules to which the functions and sorts have to obey.

[2]We do not differentiate here between these notions for the sake of simplicity.

```
data module Signal-Name
begin
  exports
    begin
      sorts
        SIGNAL-NAME
      functions
        02 :-> SIGNAL-NAME
        05 :-> SIGNAL-NAME
        08 :-> SIGNAL-NAME
        11 :-> SIGNAL-NAME
    end
end Signal-Name


data module Signal-Status
begin
  exports
    begin
      sorts
        SIGNAL-STATUS
      functions
        red   :-> SIGNAL-STATUS
        green :-> SIGNAL-STATUS
        turn  :   SIGNAL-STATUS -> SIGNAL-STATUS
    end
  equations
  [t1] turn(red)   = green
  [t2] turn(green) = red
end Signal-Status
```

The process module `Traffic-Example` is somewhat larger than the two data modules. We explain its contents by explaining all the sections top down.

As in the data modules, the process module below also has an export section. Here we find however declarations of *atoms* and *processes*. Atoms are also called *actions* or *atomic actions*. The three actions (`send-turn`, `read-turn`, `comm-turn`) take arguments from sort `SIGNAL-STATUS` and `SIGNAL-NAME` respectively.

The next section is the *import* section. If a module needs a function or process specified in another module it lists the name of that module in its import section. In this example `Traffic-Example` uses, for instance, the sort `SIGNAL-NAME` and the constants `02`, `05`, `08` and `11`. Therefore module `Signal-Name` is imported.

Not every object declared in a module needs to be declared in the export section. The process `SIGNAL : SIGNAL-STATUS # SIGNAL-NAME` is declared in a local processes section. This process is only visible in module `Traffic-Example`. The notion of locality serves to hide implementation details of a module.

In the *sets of atoms* section, one can define sets of atoms. These sets of atoms are used in combination with the `encaps` and `hide` operator. See the literature for details on hiding [BW90, M91]. The definition of process `TRAFFIC-EXAMPLE` in the *definitions* section

contains an application of `encaps`. Informally: the process `encaps(H,p)` is the process `p` where every atom of `p` that is an element of `H` is disabled. We come back on this after introduction of communication below.

Processes in PSF can communicate. Communication is modelled by having the simultaneous occurrence of two atoms result in a new atom. This simultaneous occurrence of two atoms is called synchronization. The synchronization of two atoms results in a new atom, i.e., in a communication atom, if the two atoms are allowed to communicate. In the *communication* section is specified which atoms are allowed to communicate by means of the communication function. The communication function is denoted with the infix operator '`|`'. Now the use of the `encaps` operator in the example can be explained. With the `encaps` operator communication is forced between `send-turn(ss,sn)` and `read-turn(ss,sn)`: the `encaps` operator forbids the single occurrence (execution) of `send-turn(ss,sn)` and `read-turn(ss,sn)` atoms. In other words, `send-turn(ss,sn)` and `read-turn(ss,sn)` are forced to communicate, resulting in `comm-turn(ss,sn)`.

This takes us to the last two sections. The *variable* section declares variables used in the *definitions* section. The definitions section is the most important section, here the processes are specified.

Before explaining the processes as specified in the example, it may be helpful to first introduce the process operators used. In the example four operators are used:
— the sequential composition, denoted with '`.`';
— the alternative composition, denoted with '`+`';
— the parallel composition, denoted with '`||`'.
Suppose we have the processes `p1` and `p2`. The intuition with the sequential composition `p1 . p2` is that `p2` starts after the successful termination of `p1`. The process `p1 + p2` is a process that behaves like `p1` or like `p2`. The choice between `p1` and `p2` is non-deterministic. The process `p1 || p2` is the process that performs `p1` and `p2` in parallel. This may lead to communication between atoms of `p1` and atoms of `p2` if specified in the communication function.

```
process module Traffic-Example
begin
  exports
    begin
      atoms
        send-turn, read-turn, comm-turn : SIGNAL-STATUS # SIGNAL-NAME
      processes
        TRAFFIC-EXAMPLE
        CONTROL-PROCESS
        SIGNAL : SIGNAL-NAME
    end
  imports
```

```
    Signal-Status, Signal-Name
  processes
    SIGNAL : SIGNAL-STATUS # SIGNAL-NAME
  sets of atoms
    H = {send-turn(ss,sn) | ss in SIGNAL-STATUS, sn in SIGNAL-NAME}
        +
        {read-turn(ss,sn) | ss in SIGNAL-STATUS, sn in SIGNAL-NAME}
  communications
    send-turn(ss,sn) | read-turn(ss,sn) = comm-turn(ss,sn)
      for ss in SIGNAL-STATUS, sn in SIGNAL-NAME
  variables
    signal-name   :-> SIGNAL-NAME
    signal-status :-> SIGNAL-STATUS
  definitions
    TRAFFIC-EXAMPLE =
      encaps(H,SIGNAL(02) || SIGNAL(05) ||
               SIGNAL(08) || SIGNAL(11) || CONTROL-PROCESS)

    SIGNAL(signal-name) = SIGNAL(red,signal-name)

    SIGNAL(signal-status,signal-name) =
      read-turn(turn(signal-status),signal-name) .
      SIGNAL(turn(signal-status),signal-name)

    CONTROL-PROCESS =
      (send-turn(green,02) || send-turn(green,08)) .
      (send-turn(red,02)   || send-turn(red,08)  ) .
      (send-turn(green,05) || send-turn(green,11)) .
      (send-turn(red,05)   || send-turn(red,11)  ) .
      CONTROL-PROCESS
end Traffic-Example
```

Now we will look more closely at the specific processes in the example. The process `TRAFFIC-EXAMPLE` is the parallel composition of four `SIGNALs` and one `CONTROL-PROCESS`. With the encapsulation of `H`, communication of `send-turn(red,01)` and `read-turn(red,01)`, `send-turn(green,01)` and `read-turn(green,01)` etc. is forced.
A signal with name `n` is represented by the process `SIGNAL(n)`. A signal starts with the red light on. Now, the signal `SIGNAL(red,n)` can turn green, and then turn red again etc.. But this will only happen if synchronization with the `CONTROL-PROCESS` succeeds. The `CONTROL-PROCESS`, first synchronizes with signal `02` and `08`, i.e., turning them green. Next, the control process synchronizes again with signal `02` and `08`, i.e., turning them red. A similar operation is performed with `05` and `11`, whereupon the `CONTROL-PROCESS` comes back in its initial state.

With the simulator in the PSF-Toolkit one can 'run' a PSF specification [V91]. If the example is run, the following list of atoms could be output:

```
comm-turn(green,02)
comm-turn(green,08)
comm-turn(red,08)
comm-turn(red,02)
comm-turn(green,05)
comm-turn(green,11)
comm-turn(red,05)
comm-turn(red,11)
comm-turn(green,08)
comm-turn(green,02)
.......
```

It is important to note that this could be output. What is specified is that we turn 02 and 08 green before turning them red etc.. But the order in which 02 and 08 are turned green is not specified. We therefore see both orders in the output.

### 4.2 Parameter binding

In this subsection another feature is described, namely parameter binding. Suppose we want a generic module to represent a pair of two items, no matter what the items are. To this end we introduce a data module `Pair` which has two parameters, one for the first and one for the second item in the pair. This is done in PSF by introducing a *parameters* section in a data module. In this *parameters* section we can list the parameters we need for the generic module. In the data module below we see that the two parameters are introduced in the parameter section as `ITEM1` and `ITEM2`. In the exports section we see that a sort `PAIR` and a function `pair` is introduced. The function `pair` constructs a pair of the two items mentioned, and produces thus an element of sort `PAIR`. Three other functions are introduced, namely `proj1`, `proj2` and `eq`. The function `proj1` takes a pair and produces the first item of that pair, the function `proj2` takes a pair and produces the second item of that pair.

```
data module Pair
begin
  parameters
    Item1
    begin
      sorts
        ITEM1
    end Item1,
    Item2
    begin
      sorts
        ITEM2
    end Item2
  exports
```

```
    begin
      sorts
        PAIR
      functions
        pair      : ITEM1 # ITEM2 -> PAIR
        proj1     : PAIR          -> ITEM1
        proj2     : PAIR          -> ITEM2
    end
  imports Booleans
  variables
    i1 :-> ITEM1
    i2 :-> ITEM2
  equations
  [P1] proj1(pair(i1,i2)) = i1
  [P2] proj2(pair(i1,i2)) = i2
end Pair
```

Now it is for example possible to construct pairs of which the first item is one of sort `SIGNAL-NAME` and the second item is one of sort `SIGNAL-STATUS` (see the above mentioned data modules `Signal-Name` and `Signal-Status`). In order to construct such a pair another data module has to be introduced that imports the data modules `Pair`, `Signal-Name` and `Signal-Status`, and binds the parameters of data module `Pair` to the sorts and functions exported by the data modules `Signal-Name` and `Signal-Status`. This is shown in the imports section of the data module `Signal-Name-Signal-Status-Pair` below. Here we see that the data module `Pair` is imported and that the parameters are bound by the appropriate sorts from the data modules `Signal-Name` and `Signal-Status` (which are thus also imported). At the end of the imports section we see that the sort `PAIR` is renamed by `SIGNAL-NAME-SIGNAL-STATUS-PAIR` in order to enclose, for readability, the contents of the pair in its own sort name.

```
data module Signal-Name-Signal-Status-Pair
begin
  imports
    Pair {Item1 bound by [ITEM1 -> SIGNAL-NAME]
                to Signal-Name
          Item2 bound by [ITEM2 -> SIGNAL-STATUS]
                to Signal-Status
          renamed by [PAIR -> SIGNAL-NAME-SIGNAL-STATUS-PAIR]
         }
end Signal-Name-Signal-Status-Pair
```

# 5 The specification: choices, assumptions and solutions

## 5.1 Components selected for specification

In figure 6 below the components selected for specification are shown. The data set and the control process are specified in some detail. The choices and assumptions made in specifying these components are discussed below in subsections 5.2 and 5.3 respectively.

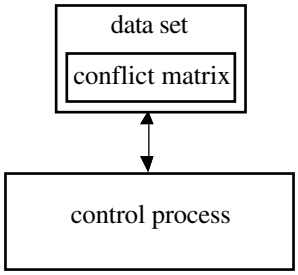


figure 6

The detection devices are modelled in the sense that in the data set is specified what detection devices exist and to which phase they obey. This enables the control process to read from a detection device some random values without really synchronizing to a detection device.

Interaction with signals is even simpler. The control process just performs actions modelling operations on the signals of a phase.

Although time is of extreme importance in traffic regulation systems, there is no 'clock-process' specified. The reason for this is that for the functionality currently specified only durations are needed. If we assume the cycle-time of the control process to have a constant duration, all other durations, e.g. `minimal-green-duration`, can easily be expressed in this cycle-time. As a result, all time-constants in the specification are discrete and relative to the implicit clock of the control process.

## 5.2 The data set

The control process specified is generic, just like the real four order regulation system. See appendix C, for an example data set. The control process assumes a data set to contain the following:

— a conflict matrix;
— a set of phases defining the initial state of the phases;
— a function giving a total ordering on priority of the phases;
— a surjective function assigning detection devices to phases;

— a function assigning a clearance duration to each phase;
— a function assigning a yellow duration to each phase;
— a function assigning a minimal green duration to each phase;
— the name of the detection device to poll first;
— the name of the detection device to poll last;
— a function giving the order in which to poll the detection devices;
— a constant specifying the duration of the regulation cycle of the control process.
Weight factors for detected traffic participants are specified, however as part of the initial state phase set.

A number of things have been left out. For instance, there is no maximum green duration, it is not possible to prohibit a third order realization of a phase, maximal allowed waiting times are not specified etc..

The control process does not check the information in the data set. It is assumed to contain correct information. Correctness of the specification in general is discussed in section 7. Here we just mention some criteria of correctness for a data set:
— the conflict matrix is symmetrical;
— a phase has no conflict with itself;
— the phase set is not empty;
— each phase has at least one detection device assigned to it;
— if a detection device of type X is assigned to a phase, this phase should contain a counter for traffic participants of type X;
— durations are positive, non-zero, integer multiples of the duration of the regulation cycle.

### 5.3 The control process

Specified are, in some detail, first order, second order realization, and third order realization of type 2, but still, what is specified is only the backbone. Numerous extra features can be found in [K90]. Some possibilities for optimization's not performed in the specification are exemplified in appendix A.1 and appendix A.4. Fourth order realizations are not considered, nor are third order realizations type 1 and 3. Because of the absence of third order type 1 realization all signals eventually turn red when there is no traffic.

The regulation is not necessarily conflict-free, but the specification offers no means to shortly delay a phase if this phase is realized together with a conflicting phase.

One can say that a second or third order phase is realized when *covered* by some other phases (not necessarily first order phases). In the specification, the notion of coverage is not made explicit, i.e., for a phase being realized in second or third order is not remembered nor explicitly calculated which phases cover the phase.

### 5.3.1 The detection devices

In every cycle of the control process, the detection devices are polled in a fixed order. Independent of the type of the detection device (push-button for pedestrians, detection field for cars) the device read out is either ‘true’ or ‘false’. So, in one cycle, a detection device can at most detect one traffic participant.

A phase contains a counter for each type of detection device assigned to it. If two or more detection devices of the same type are assigned to a phase, their read outs will be accounted for on the same counter. The type of the detection device determines in what way the device read out is accounted for. If a push button is pressed by a pedestrian the system knows there are pedestrians waiting, the system does not know how many. If a detection field for cars is activated the corresponding counter is increased by one, the system knows exactly how many cars are waiting.

The system is not able to tell how traffic participants flow away on green. When turning from green to yellow the system just assumes all detected traffic participants have flowed away!

### 5.3.2 Priority

The priority of a phase is determined by two factors. One factor is the weighed waiting time, the other the priority ordering from the data set. If two phases have the same weighed waiting time (see below), the phase with higher priority is determined by looking at the priority ordering in the data set.

The weighed waiting time of a phase is the sum of the weighed waiting time of each traffic participant type accounted by the control process for that phase. So, if a phase consists of a stream of pedestrians and a stream of cyclists, the weighed waiting time is the sum of the weighed waiting time for the pedestrians and the weighed waiting time for the cyclists. The weighed waiting time for a traffic participant type is updated each cycle of the control process. The new value is the old value plus the number of traffic participants times a weight factor. The weighed waiting time and the counter for the traffic participants are set to zero when the signal turns yellow, this is of course a simplification from our side. There is no gradual decrease during green as in the real four order system.

We will illustrate the notions above by an example. Suppose we have a phase called 01/03 in figure 3 in section 2.3. So, we combine stream 01 and 03 into one phase which we call 01/03. One of the streams is a stream of cars turning left, the other a stream of cars turning right. Suppose the weight factor for stream 01 is 1, and the weight factor for stream 03 is 2. In figure 7, the signals of phase 01/03 turn yellow in regulation cycle *i*, the weighed

waiting times and the car counters for 01 and 03 are thus set to 0. In the next regulation cycles one will see the built-up of the weighed waiting times for the streams 01 and 03, and phase 01/03. We assume the figure is self-explaining.

regulation cycles ⟶

| | | | $i$ | $i+1$ | $i+2$ | $i+3$ | $i+4$ |
|---|---|---|---|---|---|---|---|
| 01/03 | 01 | car detected | no | yes | yes | no | yes |
| | | weight factor | 1 | 1 | 1 | 1 | 1 |
| | | cars counted | 0 | 1 | 2 | 2 | 3 |
| | | weighed waiting time | 0 | 1 | 3 | 5 | 8 |
| | 03 | car detected | no | no | yes | yes | no |
| | | weight factor | 2 | 2 | 2 | 2 | 2 |
| | | cars counted | 0 | 0 | 1 | 2 | 2 |
| | | weighed waiting time | 0 | 0 | 2 | 6 | 10 |
| | | total weighed waiting time | 0 | 1 | 5 | 11 | 18 |

figure 7

In the real four order regulation system, the weighed priority is accounted once a second and the cycle time is about one tenth of a second. Furthermore, in the real system the weight factors are chosen from the interval [25..100], in our specification every natural number is fine, even zero. If one chooses zero as value for all weight factors of a phase, that phase can, only get realized third order, as in the specification the weighed waiting time is used to check whether a phase has traffic waiting.

### 5.3.3 Green duration

The estimated duration for green is taken as the maximum of the minimal green time as specified in the data set and an estimate of the requested green time. As a simplification, we take the requested green time equal to the sum of the traffic participants counted. A more realistic guess for the requested green time should take also, for instance, acceleration in consideration. Consider the situation in figure 7, if phase 01/03 is turned green during regulation cycle $i + 4$, then the requested green time is 3 + 2, i.e., the total number of cars counted. The requested green time is not tempered by taking the previous green time in consideration. So, the green time for a phase can oscillate freely. This is contrary to the real system where some levelling takes place in order to prevent strong oscillation.

For a first order realization the duration assigned for green is equal to the estimated green duration.

With second and third order realizations it is important not to frustrate phases of higher priority. This requirement leads to an upper limit for the green duration to be assigned. This upper limit is calculated as follows. Suppose a phase $p$ is a candidate for second or third order realization, and suppose we have to make sure $p$ does not frustrate the phases in a set $F$. Now, calculate for each phase in $F$ how long this phase has to wait at least before it can turn green. Take the minimum of these values. Let this be $m$. ($m$ is what is calculated by the function `spare-time` in module `Realization-Toolkit`). The upper limit is $m$ minus the clearance duration minus the yellow duration for $p$. Phase $p$ can turn green if its minimal green time is less than or equal to the upper limit. If this is the case, the green time assigned to $p$ is the minimum of the estimated green duration and the upper limit. The way a duration for green is calculated in the current specification is not optimal. An example of this can be found in example 4 in appendix A.4.

In the real four order system one specifies in the data set an extra condition for third order realization of a phase. This condition specifies for a phase $p$ which phases can cover $p$. In the specification this extra condition is not tested.

If a phase is assigned say 10 time units green, it will have precisely 10 time units, no more, no less. So, a phase is not pre-empted if it offers no traffic. And because the system is not able to detect traffic participants in the dilemma zone, green durations are not stretched. (The dilemma zone is part of the approach to the intersection. A traffic participant has a dilemma when it is hard to choose for him/her between driving on or stopping when the signals that regulates his/her stream turns yellow. A traffic participant will most probably have a dilemma if it is detected with a certain travelling speed in the dilemma zone.)

# 6 The main modules of the specification

In this section some extra attention is paid to a number of modules of the specification.

In subsection 6.1 we discuss the central data types, namely `PHASE` and `PHASE-SET`. These data types are specified in module `Phase` and module `Phase-Set`.

In subsection 6.2 and 6.3 we look at the modules specifying the most important functions on phases and sets of phases. The modules here are:

`Account-For-Detection-Operation`,

`Account-Weighed-Priorities-Operation`,

`Account-For-Time-Operation`,

`Realization-Toolkit`,

Realization-First-Order,
Realization-Second-Order,
Realization-Third-Order.

Next are the two process modules. Module Control-Process is discussed in subsection 6.4 and module Traffic-Regulation-System in subsection 6.5.

### 6.1 Module Phase and module Phase-Set

Module Phase specifies the sort PHASE. Of main importance is the function phase. One can interpret phase(name,ro,ss,we,at,rt) as a record storing status-information on the phase with name name, with realization order ro, with signal status ss, with the detected traffic participants and weighed waiting times accounted in we, with the absolute time the phase has status ss in at, and with the rest time that the phase has to keep status ss in rt. A typical example is:

```
phase(08, first-order, red,
      add(we(car,nat(^2),nat(^3),nat(^1^8)),
          add(we(pedestrian, nat(^1), nat(^1^0), nat(^5^0)),
              empty-weight-elem-set),
      nat(^1^0), nat(^2)).
```

So, this is a phase with name 08. Previously it got a first-order realization, but now is it turned red. There have been two cars detected. A car has a weight factor of 3, and the weighed waiting time is 18. There are pedestrians detected, we do not know how many (see rule [ad10] in module Account-For-Detection-Operation). Pedestrians have a weight factor of 10, and the weighed waiting time is 50. Furthermore, the phase is already red for 10 regulation cycles and it has to stay red for 2 extra cycles. This means that the clearance duration for phase 08 is 12 regulation cycles.

Apart from function phase, module Phase contains a number of functions that we think are self-explaning.

Now we turn to module Phase-Set. In module Phase-Set, module Set is bound to module Phase. This gives, after renaming, the sort PHASE-SET. This sort is of major importance. In the specification, an intersection is modelled by a set of phases and by the actual situation data set. A set of phases contains the state of the system at a certain point in time. Most of the important functions operate on a set of phases, for instance, realization-first-order from module Realization-First-Order. In module Phase-Set a number of function is introduced that give a subset of a set of phases.

### 6.2 Taking account of detection, priority and time

Before first order, second order or third order regulation can take place on a set of phases, some household activity and some simple regulation has to be done. One has to take account of detected traffic participants and one has to calculate weighed waiting time. This is performed by functions specified in module `Account-For-Detection-Operation` and `Account-Weighed-Priorities-Operation` respectively. One has to turn the signals of a phase when the granted green duration or the yellow duration has passed. This is, among other things done by functions specified in module `Account-For-Time-Operation`.

### 6.2.1 Module `Account-For-Detection-Operation`

Given a set of detection device read outs. This information says for each detection device whether there has been a detection of a traffic participant. With the function `detected-for-phase` from the actual situation data set one knows which detection devices belong to which phase. With this knowledge the function `account-for-detection` updates the traffic participant counters stored in each phase. This can be explained best by an example. Suppose the example-phase in subsection 6.1 is element of a set of phases called `P`. Suppose in the detection device read outs `DR`, we have `detection(df-car(^8),true)` and `detection(push-button-pedestrian(^1^3),true)`. Suppose in the actual situation data set we have: `detected-for-phase(df-car(^8)) = 08` and `detected-for-phase(push-button-pedestrian(^1^3)) = 08`. So, detection field 8 and pushbutton 13 belong to phase 08. Let `P'` be the result of calculating `account-for-detection(DR,P)`. Now the phase with name `08` in `P'` will be the following:

```
phase(08, first-order, red,
      add(we(car,nat(^3),nat(^3),nat(^1^8)),
          add(we(pedestrian, nat(^1), nat(^1^0), nat(^5^0)),
              empty-weight-elem-set),
      nat(^1^0), nat(^2)).
```

### 6.2.2 Module `Account-Weighed-Priorities-Operation`

The function `account-weighed-priorities` takes a set of phases as input. It updates the weighed waiting time of each 'traffic participant type' accounted for in each phase in this set of phases. To return to our running example, let `P"` be the result of `account-weighed-priorities(P')`. We show what happened to phase `08` in `P"`.

```
phase(08, first-order, red,
      add(we(car,nat(^3),nat(^3),nat(^2^7)),
          add(we(pedestrian, nat(^1), nat(^1^0), nat(^6^0)),
              empty-weight-elem-set),
      nat(^1^0), nat(^2)).
```

So, the weight waiting time of the cars is increased by 9 (3 times 3) and the weight waiting time of the pedestrians is increased by 10 (1 times 10).

### 6.2.3 Module `Account-For-Time-Operation`

The function `account-for-time` has the same structure of specification as the functions `account-for-detection` and `account-weighed-priorities`: it maps an operation on each phase in the set of phases that forms its input. On each phase `phase(n,ro,ss,we,at,rt)` it essentially does the following. If `rt` is greater than the `duration-traffic-regulation-cycle`, then `rt` is decremented. When `rt` equals the `duration-traffic-regulation-cycle` the signal status may change. If the signal status of a phase is `red` it just stays `red` and `rt` is decremented, however `rt` will not drop below `0`, but if the signal status is green or yellow, then the signal changes to yellow or red respectively.

One will notice that there is also a counter incremented, namely the absolute time `at`. This has at the moment no real purpose. In the future it may, for instance, be used to signal unacceptable red durations. To return to our example, let `P"'` be the result of `account-for-time(P")`. The phase with name `08` will now look like the following.

```
phase(08, first-order, red,
      add(we(car,nat(^3),nat(^3),nat(^2^7)),
          add(we(pedestrian, nat(^1), nat(^1^0), nat(^6^0)),
              empty-weight-elem-set),
      nat(^1^1), nat(^1)).
```

So, the rest time is decremented and the absolute time incremented.

## 6.3 First, second and third order realization

Now we come to the modules that specify the functions that take care of turning the signal status of phases to green. The functions of main interest are `realization-first-order`, `realization-second-order` and `realization-third-order`. Function that are used in the specification of these functions are given in module `Realization-Toolkit`. We will look at these functions first.

### 6.3.1 Module `Realization-Toolkit`

This module exports six functions that are used by the 'realization' functions. Below we will discuss them one by one.

Function `estimate-green-duration` is presumably clear by its name. It is only worth to mention the use of the function `total-number-of-tp-counted`. This function returns the total number of traffic participants counted, and this amount is used as an estimate of the demand for green. This has already been discussed in subsection 5.3.3.

The function named `highest-priority` selects the phase with highest priority from a non-empty set of phases.

The function `conflict` returns `true` when its first argument is in conflict with a phase in its second argument, and otherwise it returns `false`.

The function `duration-to-clearance` calculates how long it takes for a phase to get to the cleared state. (See subsection 2.3; a phase is cleared if it has its signals red for a period longer than the clearance duration.)

The function `spare-time` is used only when deciding on second or third order realization of a phase. Suppose phase `f` can get a second order (third order) realization if it does not frustrate any of the phases in set of phases `P`. That is, each phase in `P` is cleared and has higher priority than `f`, but can not be realized for some reason. Now calculate for each phase in `P` the time it takes at least before the phase can turn green. Take the smallest of these durations, call it `spare-time`. For phase `f`, to get a second order (third order) realization it is obligatory that `spare-time` is greater than or equal to `minimal-green-duration(f) + yellow-duration(f) + clearance-duration(f)`. See rules `[so5]` to `[so7]` in module `Realization-Second-Order` for an application of the function `spare-time`. In rule `[so5]` we have:

```
spare-time(conflict-in(name(f),not-used),select-not-cleared(fs)).
```

Here `not-used` is set of phases that are requesting realization and have higher priority than `f`, but can not be realized now. With `conflict-in(name(f),not-used)` one selects those

phases in `not-used` that are in conflict with `f`. Why can the phases in `conflict-in(name(f),not-used)` not be realized? Well, they have to wait for some phases that are still not cleared. These phases are selected from `fs` with `select-not-cleared(fs)`.

### 6.3.2 Module `Realization-First-Order`

To explain the function `realization-first-order` it is probably useful to investigate rule `[f01]` in module `Realization-First-Order`. This rule is copied below.

```
[f01] realization-first-order(fs)
      = realization-first-order-1(
          diff(fs,select-traffic-waiting(
                    select-rest-time-zero(
                      select-red(fs)))),
          select-traffic-waiting(
            select-rest-time-zero(
              select-red(fs))),
          empty-phase-set)
```

Function `realization-first-order-1` is used here to do the work. This function has three arguments, all three are sets of phases. It will always be true that the union of these three sets of phases is the set of phase `fs`, except for updates in the status of some phases, i.e., those that are turned green. The first set contains the phases in `fs` that are *not* potential candidates for first order realization. The second set contains the phases in `fs` that are potential candidates for first order realization; these phases are cleared (see section 2.3) and have traffic waiting, so they are requesting realization. Initially the third and last set is empty. Later on it will contain those phases in the second set (and removed from it) that didn't make it, i.e., that didn't get a first order realization after all.

What happens now is the following. Check if there are still candidates left (rule `[fo2]`, `[fo3]`). If not, your done. If there are candidates, select the one with the highest priority. Then decide (rule `[fo4]`, `[fo5]`) if this phase has no direct conflict. This test is performed by the function `direct-conflict` (rule `[fo6]`). If there is a conflict, then this phase won't make it; it is removed from the set of candidates and added to the set of neglected phases called `not-used`. If there is no conflict the phase is turned green by `turn-green-first-order` (rule `[tg1]`).

### 6.3.3. Module `Realization-Second-Order`

Function `realization-second-order` has a similar specification as `realization-first-order`. But with a first order realization it is enough to check only on a direct conflict with `direct-conflict`. With a second order realization it is important that there is no direct conflict and no frustration of other phases (see the discussion on `spare-time` in subsection 6.3.1). 'Spare time' is what is extra checked upon in rules `[so6]` and `[so7]`.

### 6.3.4 Module `Realization-Third-Order`

Function `realization-third-order` has a specification similar to `realization-second-order`, we kept the numbering of the rules the 'same' for the 'same' rules. Extra rules are `[to1b]`, `[to5a]` and `[to5b]`. These extra rules stem from two extra tests that are added.

The first test added is in the beginning, in rule `[to1a]` and `[to1b]`. The test here, is on the existence of a phase with green signals. If there is not such a phase, then there is no possibility to realize a phase third order *type 2* [K90, III 3.2]. Rules `[to1a]` and `[to1b]` can however easily be extended by adding rules for third order realization *type 1*. If the test in `[to1a]` and `[to1b]` is left out and there is no traffic, the system will still not realize phases third order. This is because we then have in `[to5a]`:

```
time-left(duration-to-clearance(select-green(fs))) =
time-left(duration-to-clearance(empty-phase-set)) =
time-left(empty-natural-set) = 0.
```

So, there is still 'no time left' to realize a phase third order.

The green duration calculated and assigned to a phase in rules `[to5a]` to `[to9]` is far from optimal. The idea is to assign only the time that is strictly needed and not more than that. A result of non-optimal regulation is shown in example 4 in appendix A.4.

The second test is found in rules `[to5a]` and `[to5b]`. This test is needed because `spare-time` is only at stake when there is the danger of frustration of a phase, and this can only the happen when `overrun` is not empty.

It is worth the note why this situation can not occur, normally, when using `realization-second-order`. In normal use of `realization-second-order` on a set of phases `P`, the following should hold (this is however not tested upon in the specification):

`realization-first-order(P) = P.` (*)

Let `tw-zero-red` be the initial, non-empty set of potential candidates for second order realization (the second argument of `realization-second-order-1`). By *, the following holds for each phase in `tw-zero-red`:

the phase has a conflict with a phase with higher priority that is also requesting realization, or the phase has a direct conflict. (**)

(See the rules for realization 1.2 and 1.3 in subsection 2.3.) Suppose we are in the initial situation. As we know `tw-zero-red` is not empty, so `realization-second-order-2` is called. Let `f` be the phase with highest priority in `tw-zero-red`. Phase `f` must have a direct conflict, otherwise it would have been realized first order. So, rule `[so5]` is not used. Now suppose that somewhere in the execution of `realization-second-order` the next phase in priority in `tw-zero-red` is phase `f`, and suppose also that `f` has no direct conflict. By ** we obtain the following: since `f` has no direct conflict now, it must have a conflict with a phase `f'` that is also requesting realization and has a higher priority. And again, since `f` has no direct conflict now, `f'` did not get a second order realization. So, `f'` is in `not-used`. Thus `conflicts-in(name(f),not-used)` is not empty.

Note that `overrun` in `[to5b]` does not contain phases from `not-used`. This is because phases in `not-used` have no traffic waiting, and therefore can hardly be frustrated. Note also that it is important that the following holds when applying `realization-third-order` on set of phases `P`:

`realization-first-order(realization-second-order(P)) = P.`

If this condition is not fulfilled, then we can have `spare-time(..) = 0` in `[to5b]`. Although the results are not undefined, it is not a proper use of `realization-third-order`.

### 6.4 Module `Control-Process`

In this module the process `CONTROL-PROCESS` is specified. In the first rule `CONTROL-PROCESS` is initialized with the initial status, that is the set of phases in their initial state as specified in the `Actual-Situation-Dataset`. The process `Control-Process(status)` essentially performs a loop, where in each (traffic regulation) cycle, detection devices are read, next a new status is calculated, and finally the signals are changed.

The process `Poll-Detection-Devices` polls all detection devices, in the order specified in the actual situation data set. This order may be irrelevant from a specification point of view, it is only given because it is given in the real system.

The actual calculation of the new status, is done by nesting the application of the functions discussed above:

```
realization-third-order(
  realization-second-order(
    realization-first-order(
      account-for-time(
        account-weighed-priorities(
          account-for-detection(
            add(detection(detector,b),ds),status))))))
```

Because there are no signals specified, the changing of the signals is not fully specified, there is just an atom, namely `change-signal`. But here the specification can easily be extended.

### 6.5 Module `Traffic-Regulation-System`

In this module the main process `TRS` (traffic regulation system) is specified. But the most important point is the binding of module `Control-Process` to `Actual-Situation-Dataset`.

## 7 Comments and future work

### 7.1 Comments

We think we succeeded in specifying a non-trivial part of *a* four order regulation system in PSF. That is, there is a distance between the specification, and the real four order regulation system. This distance stems from the material on which the specification is mainly based: the discussions with Jan Kroone and Henk Spronk, and the reference manual for UAP-26 [K90]. We had a close look at the outside of the system, but could only guess about the inside. Thus, if we say, we succeeded in specifying a non-trivial part of a four order regulation system in PSF, this means, we caught in PSF, part of a look-alike, without having trouble to express things in PSF. For instance, the modelling of time turned out not to be a problem, this comes from the fact we could use relative discrete durations. We feel confident that we can add features without running out of expressive power. However, we are not satisfied with the data-oriented approach to the problem, a process oriented approach would be more interesting. We return to this in subsection 7.2.

So, it seems reasonable to suggest that PSF could serve as a tool in the design and analysis phase of traffic regulation systems like the one we studied. The emphasis with the use of PSF in the design phase of a such system should be on structure, main functions and interaction of the components of this system. PSF is not especially geared to modelling real-time aspects, as a result an extra specification step would be needed before implementation

could commence on basis of say the current specification. In this step a PSF specification should be annotated with real-time demands.

We have the impression that currently Nederland Haarlem delivers quality software and offers a stable environment for engineers to work in. Products are developed step by step in a carefully and evolutionary way. Therefore, there is in our opinion no urgent reason to introduce a formal specification language now. But a formal specification language could be very useful when Nederland Haarlem radically changes the design of its software, or starts the development of a product for a market that they don't know so extremely well as the market of the traffic management systems.

The primary gain with the use of a formal specification language as PSF is found in the quality of the design and good technical documentation. This leads, as a result, to a smoother implementation phase, test phase, and improved maintainability. We have to stress that formal specification as a technique, does not set a new goal. It can be seen as the next step in the evolution of structured system design techniques (for example described in [Yo89]). Formal specifications are by no way guarantied error free. The specification at hand will probably contain quite some errors. But the specification language forces us to be clear and concise and thus think better on the design. As a result the design is less error prone. The formal specification language is a tool supporting the mental effort during design as Cohen, Harwood and Jackson once put it [CHJ86]. In our opinion these features of formal specification languages should do. If one can simulate a specification, this gives of course extra confidence in the design (but only a crash proves something). However, not every formal specification language comes with a simulator. And as final resort, one can, in principle, prove assertions on the systems specified. There is of course a trade-off between the confidence one seeks and the amount of effort needed to reach that level of confidence. Such a verification effort can be extremely difficult. The interesting point is that it stresses the inherent conciseness of formal specifications, and that all this can be done 'with pen and paper', i.e., before the system is implemented.

What about the tool support of PSF. The simulator does have some problems with the specification: it takes minutes to perform one regulation cycle on twelve phases. Probably the specification can be enhanced with respect to simulation, but we expect this will decrease the readability of the specification. A more structural approach would be to speed-up the term reduction machine of the PSF toolkit. To ease industrial acceptance of PSF the toolkit should contain some graphical oriented tools. A simple tool that could show the modular structure of the specification graphically and allows the user to browse through the specification could be a start. Furthermore a pretty printer that removes a lot of the keywords, highlight others, and

turns `sum(a in A,...)` into $\sum_{a \in A}(...)$ would be nice. In other words, there is a need for presentation tools of formal specifications. More advanced is a graphical front-end placed on top of the simulator. Steps in a process trace produced by the simulator could then be represented by state changes of graphical objects (specified in some additional language), at least for traffic regulation this would be very convincing!

As we are not in the field of traffic and transportation science, we can not say anything specific about efficiency, optimal solutions, demanded behaviour etc., nor about the amount and type of intersections that can be caught by the data sets we specify. The interesting questions and solutions here can only be given by people experienced in this field. Once we have formulated (with help) the qualitative demands on the system (correctness criteria), the techniques from the field of specification verification can be used to check them (provided it does not turn out to be terrible difficult). Suppose we have defined a notion of correctness for the following objects:

— correctness of the status, i.e., the set phases;
— correctness of the read out of a detection device;
— correctness of the scheduling performed in one cycle of the control process;
— correctness of the data set.

Now we could define correctness of a generic regulation system by:

> A generic regulation system is correct if for every correct data set the generic regulation system bound to this data set transforms a correct status under input of correct detection device read out to a correct status. Furthermore, the system is fair (every phase with traffic waiting gets its turn) and the system shows progress.

## 7.2 Future work

Follow-up on this project can take several directions:

— it might be interesting to do the specification anew, but now process oriented;
— one can extend the specification: processes for signals, detection devices and conflict monitor, interactions with other regulation systems;
— one could detail the current specification: fourth order realization, refine the conditions in the data set etc..
— one could specify more precisely the correctness criteria and investigate what can reasonably be verified;
— The specification of the data type `PHASE-SET` should be enhanced. This data type is better specified by tables with phase names as keys and the other arguments to the current function `phase` together as value. So, sort `PHASE-SET` contains a lot of garbage, i.e., not meaningful objects.

— we have not proved the completeness of the term reduction system that is inherent in the data modules.

Probably the most interesting direction to take is the first. This is for several reasons. First, it could lead to an interesting specification where the real strength of PSF, the process directed part of the language, can be put to work. Second, it may result in an interesting protocol. And last, it may lead to a new approach to specify 'schedulers' like the control process. The idea would be to represent each phase by a process. Phases would fight for realization. In order to find out who's turn it is, the phases will communicate with other phases, comparing priorities etc..

## References


[B91] Brunekreef J.J., A formal specification of three Sliding Window Protocols, Report P9102, Programming Research Group, Faculty of Mathematics and Computer Science, University of Amsterdam, 1991.

[BW90] Baeten J.C.M., Weijland W.P., Process Algebra, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[BHK89] Bergstra J.A., Heering J., Klint P. eds., Algebraic Specification, ACM Press Frontier Series, Addison-Wesley, 1989.

[CHJ86] Cohen B., Harwood W.T., Jackson M.I., The specification of complex systems, Addison-Wesley, 1986.

[GP90] Grootte J.F., Ponse A., The syntax and semantics of μCRL, Report CS-R9076, CWI, Amsterdam, 1990.

[GP91] Grootte J.F., Ponse A., μCRL: A base for analysing processes with data, In E. Best and G. Rozenberg, eds., Proceedings third workshop on concurrency and compositionality, Goslar, GMD-studien Nr. 191, pages 125-130, Universität Hildesheim, 1991.

[HP92] Hillebrand J.A., Ponse A., An algebraic specification of a Model Factory, part II, Report P9214, Programming Research Group, Faculty of Mathematics and Computer Science, University of Amsterdam, 1993.

[K90] Kroone J.J.M.T., UAP–26 Universele applicatieprogramma's voor de FR34 en FR80, B.V. Nederland Haarlem Verkeerssystemen, 1990.

[KVW93] Klusener S., Vlijmen S.F.M. van, Waveren A. van, Service Independent Building Blocks-I; Concepts, Examples and Formal Specifications, Report

P9310, Programming Research Group, Faculty of Mathematics and Computer Science, University of Amsterdam, 1993.

[MM90] Mauw S., Max Gy., A formal specification of the Ethernet protocol, Report P9007, Programming Research Group, Faculty of Mathematics and Computer Science, University of Amsterdam, 1990.

[MV89] Mauw S., Veltink G.J., An introduction to PSFd, in: Proc. International Joint Conference on Theory and Practice of Software Development, TAPSOFT '89, (J. Diaz, F. Orejas, eds.) LNCS 352, pp. 272-285, Springer Verlag, 1989.

[MV90] Mauw S., Veltink G.J., A process specification formalism, Fundamenta Informaticae XIII, pp. 85-139, 1990.

[M91] Mauw S., PSF a process specification formalism, PhD Thesis, University of Amsterdam, 1991.

[O92] Overkamp A., Schuppen J.H. van, Centre of Mathematics and Computer Science, Amsterdam, personal communication.

[PV93] Ponse A., Verschuren J.A., An algebraic specification of a Model Factory part III, Report P9303, Programming Research Group, Faculty of Mathematics and Computer Science, University of Amsterdam, 1993.

[W93] Wamel J.J. van, A Library for PSF, Report P9301, Programming Research Group, Faculty of Mathematics and Computer Science, University of Amsterdam, 1993.

[SS93] Schoneveld S., Stins S., Specification of a Bank Account with Process Algebra, Report P9307, Programming Research Group, Faculty of Mathematics and Computer Science, University of Amsterdam, 1993.

[W91] Wamel J.J. van, A formal specification of three simple protocols, Report P9110, Programming Research Group, Faculty of Mathematics and Computer Science, University of Amsterdam, 1991.

[W93] Wamel J.J. van, A Library for PSF, Report P9301, Programming Research Group, Faculty of Mathematics and Computer Science, University of Amsterdam, 1993.

[Yo89] Yourdon E., Modern structured analysis, Prentice Hall, 1989.

# A Example simulations of the specification

## A.1 Example 1

In the example in section 3 we stopped after one regulation cycle. Here we discuss what could happen next. As we will see, the system as specified now is not optimal because of the simplifications we made. After the first regulation cycle, discussed in the example in section 3, we come to the situation depicted in figure A.1 below (some traffic participants already left).

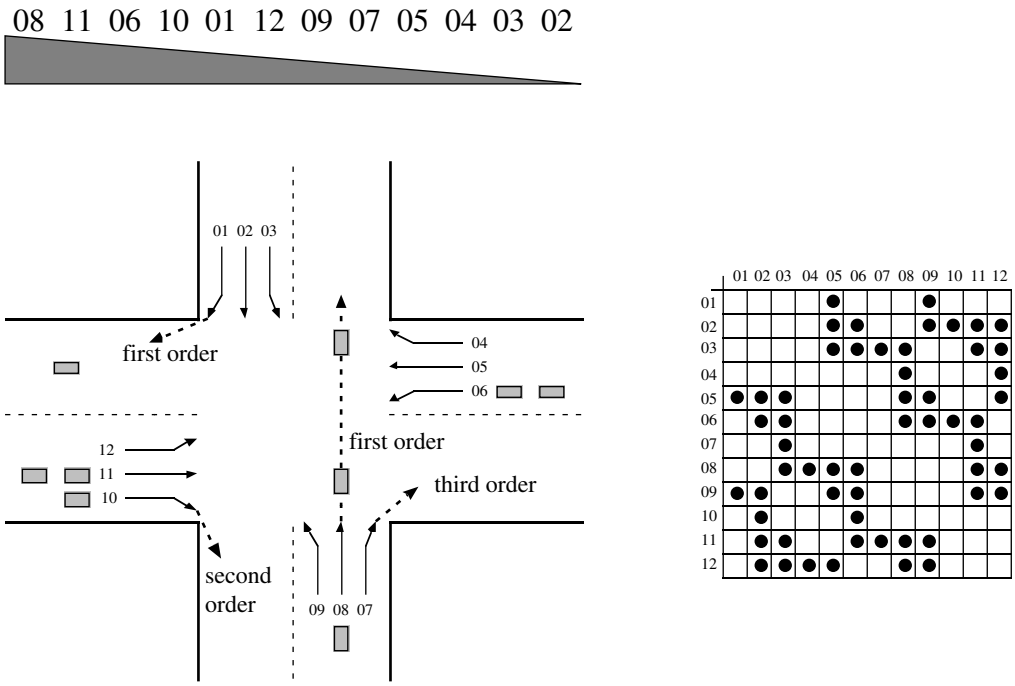


figure A.1.

The point now is what happens when phase 08 is turned red after a while. When we started regulating, phase 06 had a higher priority than phase 11. But this need not be the case some cycles later. So, it can happen that when phase 08 is turned red, phase 11 has grown in priority above phase 06 (and all other phases). In that case phase 11 will be turned green first order. As a result, there is in the end no reason to turn phase 10 red together with phase 08. In order to avoid a situation like this one, a system has to estimate what probably will happen, something that is clearly not done in the simplified system we specified.

## A.2 Example 2

Suppose we have the situation depicted below in figure A.2.1. The ordering of the phases depicted is now the (absolute) priority ordering from the actual situation dataset (to stress the difference the triangle is filled black instead of gray as in the previous examples). The conflict matrix is also depicted in figure A.2.1 below. The other parameters in the actual situation dataset are:

— all clearance durations, yellow durations and minimal green durations are set equal to the cycle time;

— all weight factors are set to one, except the weight factor of phase 11 which is set to 6.

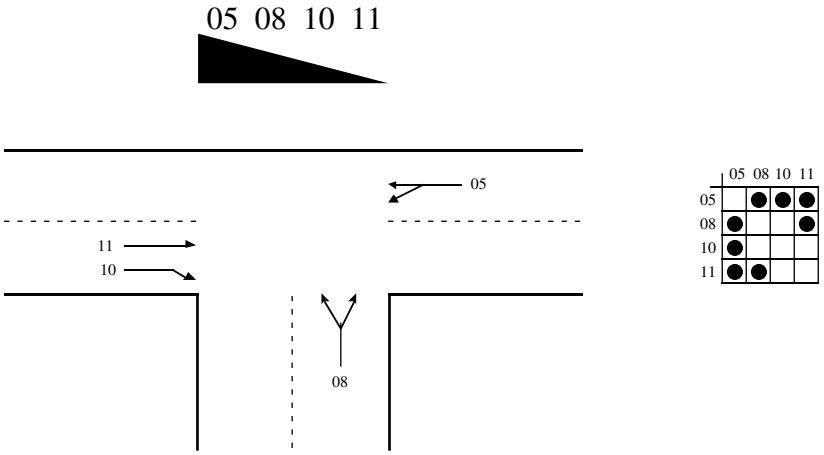


figure A.2.1

Before looking at the trace, we first explain by means of figure A.2.2 below, the meaning of the various objects used in figure A.2.3.

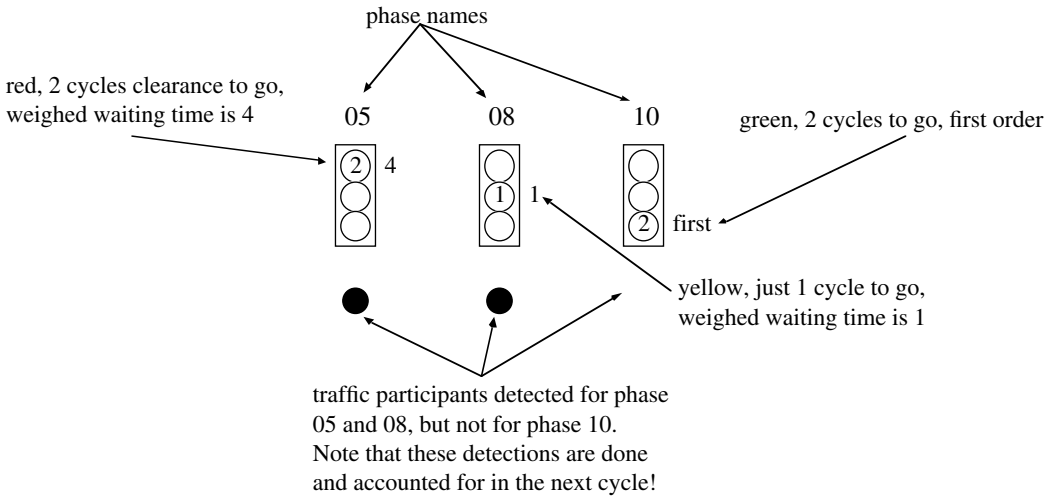


figure A.2.2

So, phase 11 has a higher weight factor than the other phases. As will be shown in the graphically depicted trace in figure A.2.3 below, the priority of phase 11 can grow faster than the priority of the other phases. As a result phase 11 can overrun other phases in priority. Phase 08 detects a traffic participants before phase 11 does detect something. So the weighed waiting time of phase 08 starts to grow when the weighed waiting time of phase 11 is still zero. But then phase 11 detects a traffic participant and its weighed waiting time jumps in one go to the same level as that of phase 08. In the next cycle the weighed waiting time of phase 08 grows to 10 and that of phase 11 grows to 12. So, phase 11 gets a first order realization before phase 08.

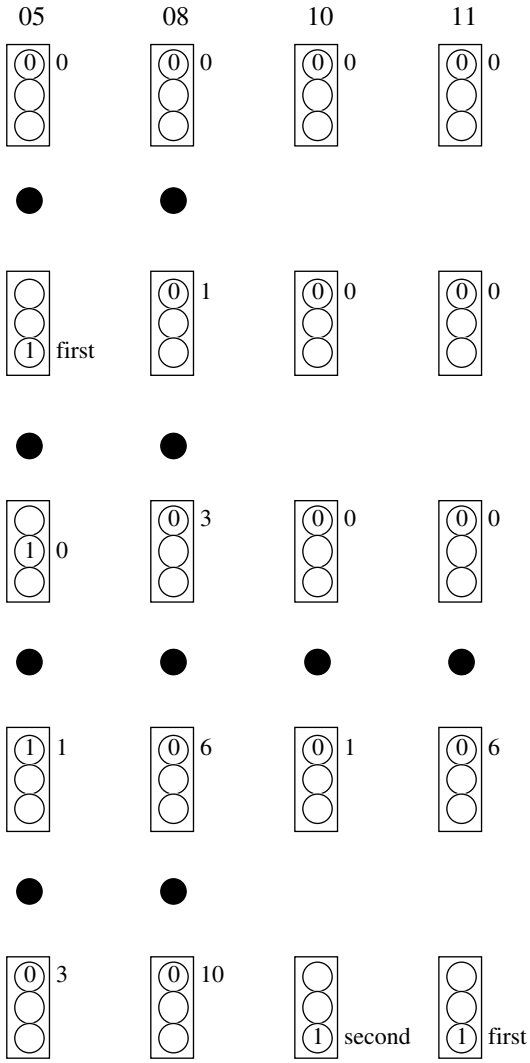


figure A.2.3

## A.3 Example 3

Suppose we have the situation depicted below in figure A.3.1. The (absolute) priority ordering in black again (as in example 2). The conflict matrix is also depicted in figure A.3.1 below. The other parameters in the Actual-Situation-Dataset are:

— all clearance durations, yellow durations and minimal green durations are set equal to the cycle time;

— all weight factors are set to one.

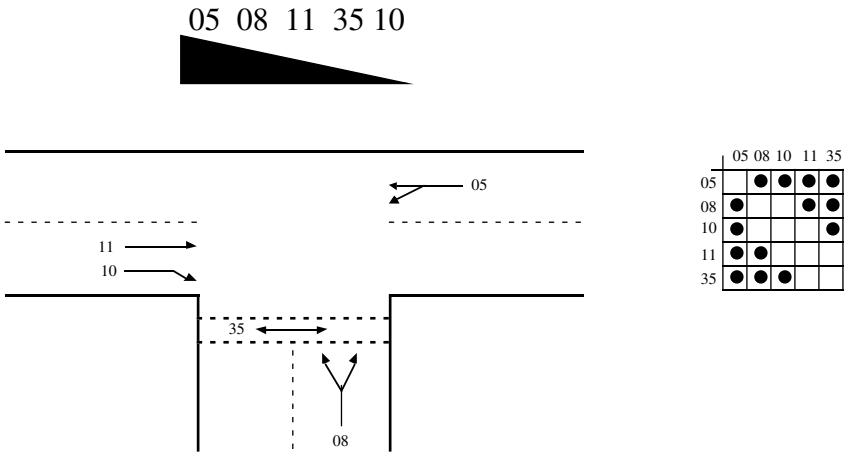


figure A.3.1

Phase 35 models a pedestrian crossing. The purpose of this example is to show that a second or third order realization need not necessarily start at the same moment as a certain first order realization. Below we show this by having phase 11 green for some time already, when phase 10 gets a second order realization, still under cover of phase 11.

The order in which to read the figure on the next page is: first the left column, and then the right column.

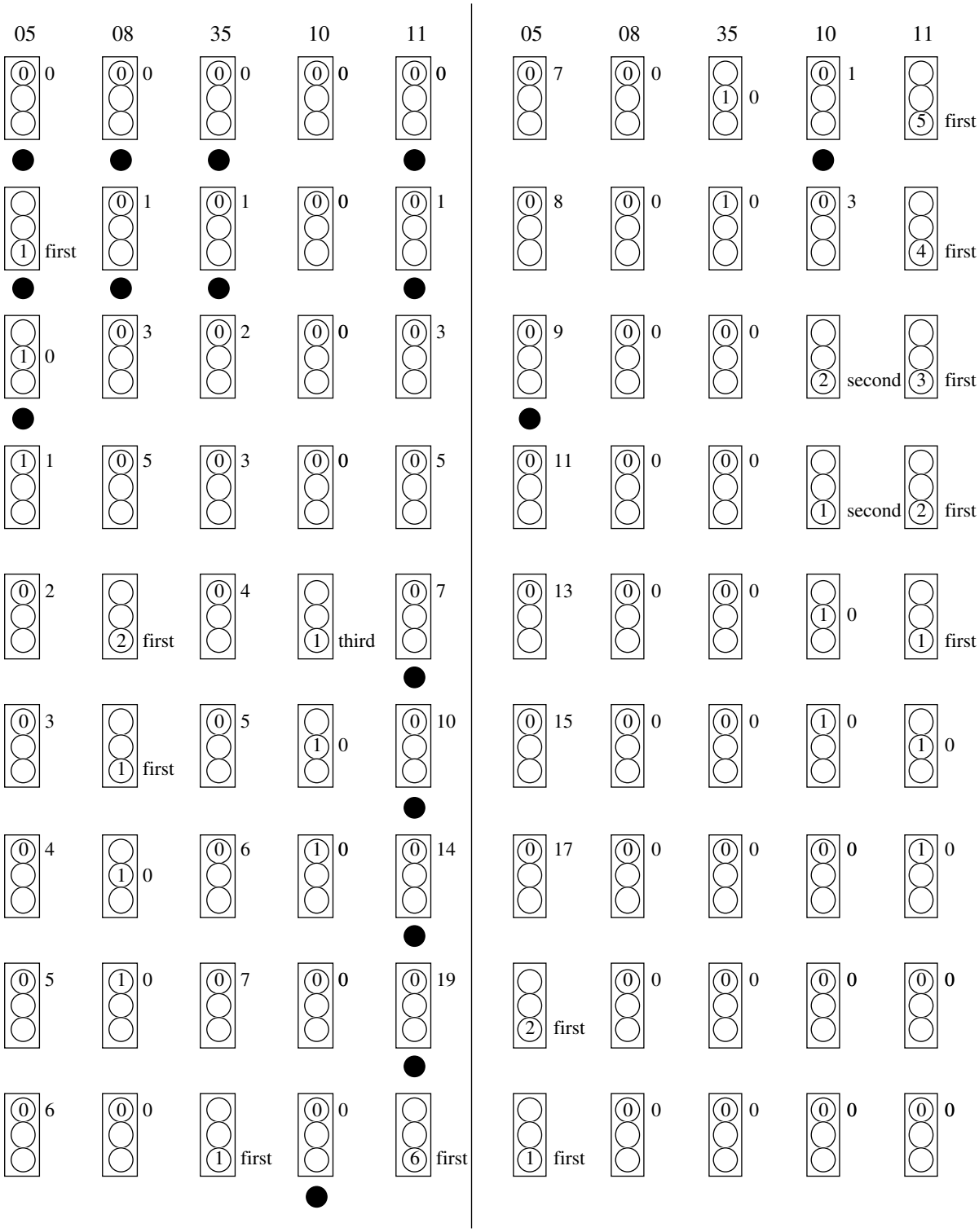


figure A.3.2

## A.4 Example 4

In this example a trace is given of the specification bound to `Actual-Situation-Dataset` in appendix C. The intersection is one that is already used before, namely in section 3 and in example 1 in appendix A.1. First, this example serves to give the reader a trace of the regulation of a more complex intersection than is given in example 2 and 3. Second, the trace shows clearly points where the regulation, as we have specified it, is not optimal. There are probably a number of other points, where the regulation, as we specified it, is not optimal. As it is not the aim of this paper to investigate the quantitative behaviour of the system specified, we only discuss possible optimization's here in order to help the reader to better understand this trace, the specification and the choices made in the specification.

In the figure on the next page, four boxes are drawn in dashed lines and marked with a number in the upper right corner.

In the box marked with '1' we have phase 04 being realized second order. The interesting point here is that phase 04 is covered here not by a first order phase but by a second order phase, namely phase 05.

In the box marked with '2', we see that phase 07 is green for 2 regulation cycles, in third order. But, we can safely turn phase 07 green for three regulation cycles: phase 07 is completely covered by phase 06, and phase 06 is green for three regulation cycles. We have the same situation in the box marked with '4'. Here we have that phase 07 is again complete covered by a phase, now by phase 08.

In the box marked with '3' another type of optimization is possible. Here phase 07 can not be covered by phase 06: there is no spare time (see subsections 5.3.3 and 6.3.1). But, it is certain that in the next regulation cycle phase 07 will be covered by phase 08. Hence, phase 07 could be turned green already. We will now show that phase 08 will turn green first order in the next regulation cycle. If all phases except phase 02, 08 detect a car in the next cycle, phase 02 will still have the highest priority of all phases and thus get a first order realization. Because in this situation the weighed waiting time of the phases (except phase 01) would be 28, 11, 4, 1, 3, 1, 20, 17, 26, 18, 19. The only phases that can possibly turn green are the phases that have no conflict with phase 01 and phase 02. These are the phases 03, 04, 07 and 08. Phase 08 would have a higher priority (20) than 03, 04 and 07 (11, 4 and 1 respectively). Furthermore, the only phase with a higher priority than phase 08 would be phase 10. Phase 10 would have priority 26, whereas phase 08 would have priority 20. But phase 08 has no conflict with phase 10. Hence, phase 08 will be realized first order in the next regulation cycle.

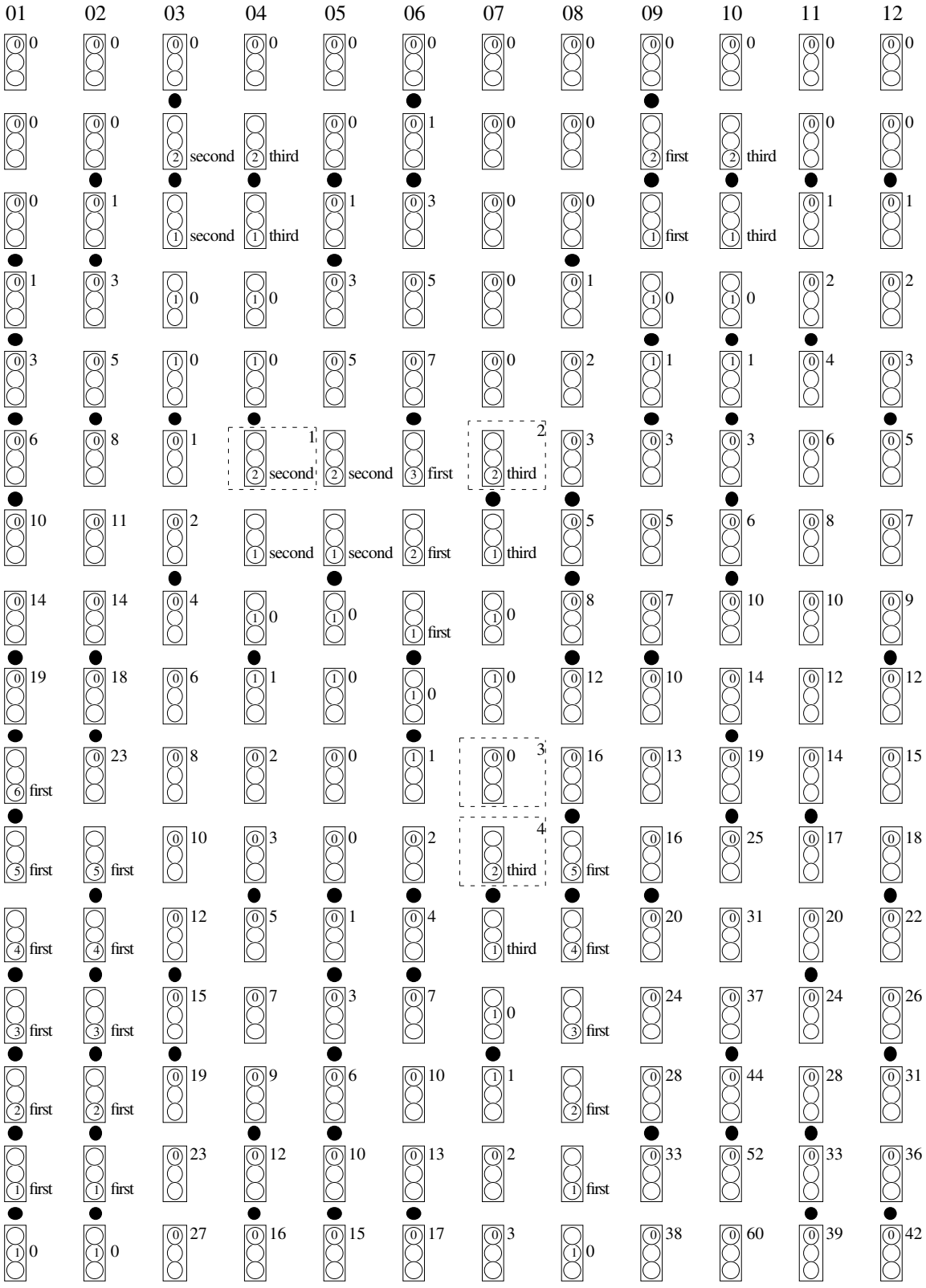


figure A.4.1

# B The specification

The specification is found in this appendix and the next. The import structure of the most important modules is depicted in figure B.1 below.

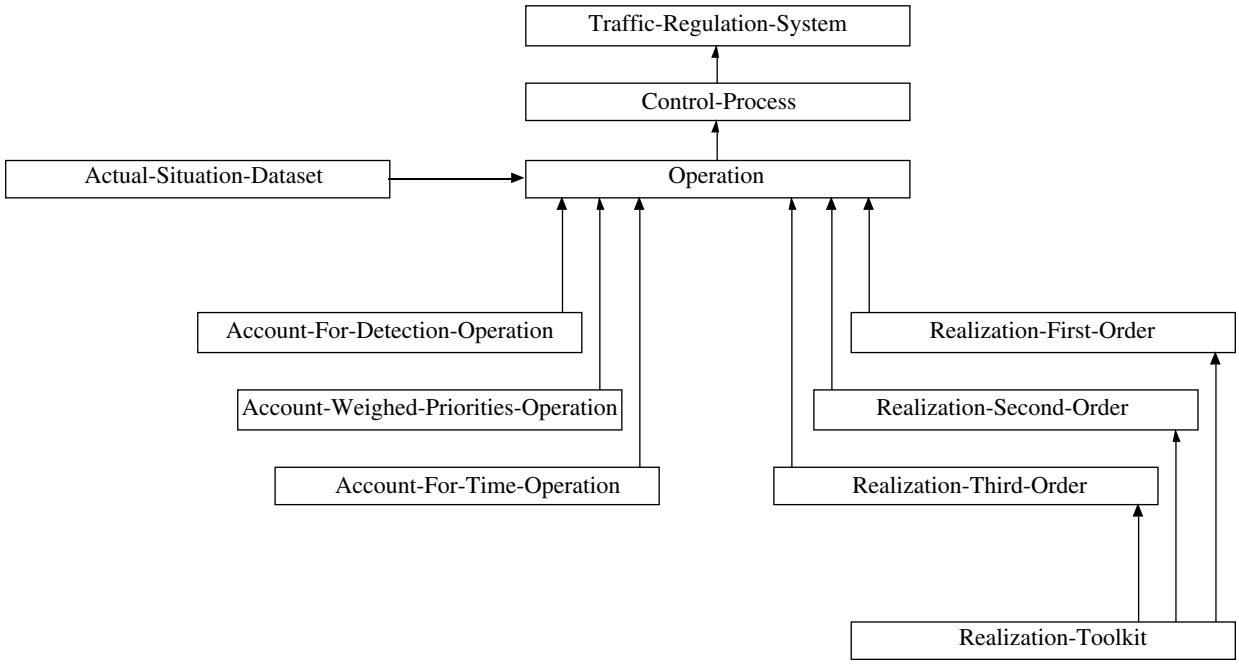


figure B.1

Module `Actual-Situation-Dataset` is found in appendix C. The other modules in figure B.1 are listed below, however at the tail of the appendix. The listing starts with a number of small modules. These are not depicted in figure B.1 in order not to blur the main structure. Note that in the listing below modules from the PSF library, e.g., `Numerical-Library`, are not listed, for these modules we refer the reader to [W93].

```
data module Detection-Device
begin
  exports
    begin
      sorts
        DETECTION-DEVICE
      functions
        push-button-pedestrian: NATURAL -> DETECTION-DEVICE
        push-button-cyclist   : NATURAL -> DETECTION-DEVICE
        df-car                : NATURAL -> DETECTION-DEVICE
        df-public-transport   : NATURAL -> DETECTION-DEVICE
        df-assistance-service : NATURAL -> DETECTION-DEVICE
        eq                    : DETECTION-DEVICE # DETECTION-DEVICE -> BOOLEAN
    end
  imports
    Numerical-Library
  functions
    type : DETECTION-DEVICE -> NATURAL
    ord  : DETECTION-DEVICE -> NATURAL
  variables
    n       :-> NATURAL
    dd, dd' :-> DETECTION-DEVICE
  equations
  [eq1]  type(push-button-pedestrian(n)) = nat(^0)
  [eq2]  type(push-button-cyclist(n))    = nat(^1)
  [eq3]  type(df-car(n))                 = nat(^2)
  [eq4]  type(df-public-transport(n))    = nat(^3)
  [eq5]  type(df-assistance-service(n))  = nat(^4)
  [eq6]  ord(push-button-pedestrian(n))  = n
  [eq7]  ord(push-button-cyclist(n))     = n
  [eq8]  ord(df-car(n))                  = n
  [eq9]  ord(df-public-transport(n))     = n
  [eq10] ord(df-assistance-service(n))   = n
  [eq11] eq(dd,dd') = and(eq(type(dd),type(dd')),eq(ord(dd),ord(dd')))
  [eq12] eq(dd,dd') = eq(dd',dd)
end Detection-Device

data module Set
begin
  parameters Item
    begin
      sorts
        ITEM
      functions
        eq      : ITEM # ITEM -> BOOLEAN
    end Item
  exports
    begin
      sorts
        SET
      functions
        empty   :             -> SET
        add     : ITEM # SET -> SET
        is-in   : ITEM # SET -> BOOLEAN
        rem     : ITEM # SET -> SET
        union   : SET  # SET -> SET
        isect   : SET  # SET -> SET
        diff    : SET  # SET -> SET
        subset  : SET  # SET -> BOOLEAN
```

```
      card    : SET        -> NATURAL
      eq      : SET  # SET -> BOOLEAN
    end
  imports
    Numerical-Library
  variables
    i, i1, i2 :-> ITEM
    s, s', s'':-> SET
  equations
  [A1]  add(i, add(i, s)) = s
  [A2]  add(i1, add(i2, s)) = add(i2, add(i1, s))

  [ii1] is-in(i, empty) = false
  [ii2] is-in(i1, add(i2, s)) = or(eq(i1, i2), is-in(i1, s))

  [R1]  rem(i, empty) = empty
  [R2]  rem(i, add(i, s)) = rem(i, s)
  [R3]  eq(i1, i2) = false ==> rem(i1, add(i2, s)) = add(i2, rem(i1, s))

  [U1]  union(empty, s) = s
  [U2]  union(add(i, s), s') = add(i, union(s, s'))
  [U3]  union(s, s') = union(s', s)
  [U4]  union(s, union(s', s'')) = union(union(s, s'), s'')

  [i1]  isect(empty, s) = empty
  [i2]  is-in(i, s') = true ==> isect(add(i, s), s') = add(i, isect(s, s'))
  [i3]  is-in(i, s') = false ==> isect(add(i, s), s') = isect(s, s')
  [i4]  isect(s, s') = isect(s', s)
  [i5]  isect(s, isect(s', s'')) = isect(isect(s, s'), s'')

  [D1]  diff(s, empty) = s
  [D2]  diff(s, add(i, s')) = rem(i, diff(s, s'))

  [S1]  subset(empty, s) = true
  [S2]  subset(add(i, s), s') = and(is-in(i, s'), subset(s, s'))

  [C1]  card(empty) = nat(^0)
  [C2]  card(add(i, s)) = nat(^1) + card(rem(i, s))

  [eq1] eq(s, s') = and(subset(s, s'), subset(s', s))
  [eq2] eq(s, s') = eq(s', s)
end Set

data module Natural-Set
begin
  exports
    begin
      functions
        min : NATURAL # NATURAL-SET -> NATURAL
        max : NATURAL # NATURAL-SET -> NATURAL
    end
  imports
    Set {Item bound by
           [ITEM -> NATURAL]
         to Numerical-Library
         renamed by
           [SET -> NATURAL-SET,
            empty -> empty-natural-set]
        }
```

```
  variables
    n, m    :-> NATURAL
    rest-ns :-> NATURAL-SET
  equations
  [mi1] min(n,empty-natural-set) = n
  [mi2] min(m,add(n,rest-ns)) = min(m,min(n,rest-ns))

  [ma1] max(n,empty-natural-set) = n
  [ma2] max(m,add(n,rest-ns)) = max(m,max(n,rest-ns))
end Natural-Set

data module Phase-Name
begin
  exports
    begin
      sorts
        PHASE-NAME
      functions
        01   : -> PHASE-NAME
        02   : -> PHASE-NAME
        03   : -> PHASE-NAME
        04   : -> PHASE-NAME
        05   : -> PHASE-NAME
        06   : -> PHASE-NAME
        07   : -> PHASE-NAME
        08   : -> PHASE-NAME
        09   : -> PHASE-NAME
        10   : -> PHASE-NAME
        11   : -> PHASE-NAME
        12   : -> PHASE-NAME
        eq   : PHASE-NAME # PHASE-NAME -> BOOLEAN
    end
  imports
    Numerical-Library
  functions
    I : PHASE-NAME -> NATURAL
  variables
    fn, fn' :-> PHASE-NAME
  equations
  [I1]  I(01) = nat(^0)
  [I2]  I(02) = nat(^1)
  [I3]  I(03) = nat(^2)
  [I4]  I(04) = nat(^3)
  [I5]  I(05) = nat(^4)
  [I6]  I(06) = nat(^5)
  [I7]  I(07) = nat(^6)
  [I8]  I(08) = nat(^7)
  [I9]  I(09) = nat(^8)
  [I10] I(10) = nat(^9)
  [I11] I(11) = nat(^1^0)
  [I12] I(12) = nat(^1^1)
  [eq1] eq(fn,fn') = eq(I(fn),I(fn'))
  [eq2] eq(fn,fn') = eq(fn',fn)
end Phase-Name

data module Phase-Name-Set
begin
  imports
    Set {Item bound by
```

```
          [ITEM -> PHASE-NAME, eq -> eq]
        to Phase-Name
        renamed by
          [SET -> PHASE-NAME-SET,
           empty -> empty-phase-name-set]
      }
end Phase-Name-Set

data module Conflict-Matrix
begin
  exports
    begin
      sorts
        CONFLICT-MATRIX
      functions
        empty-conflict-matrix : -> CONFLICT-MATRIX
        add                   : PHASE-NAME # PHASE-NAME # CONFLICT-MATRIX
                                -> CONFLICT-MATRIX
        conflict              : PHASE-NAME # PHASE-NAME # CONFLICT-MATRIX
                                -> BOOLEAN
        conflicts-of          : PHASE-NAME # CONFLICT-MATRIX -> PHASE-NAME-SET
    end
  imports
    Phase-Name-Set
  variables
    f, f', g, g' :-> PHASE-NAME
    cm           :-> CONFLICT-MATRIX
  equations
  [cm1] add(f,f,cm) = cm
  [cm2] add(f,f',add(f,f',cm)) = add(f,f',cm)
  [cm3] add(f,f',add(g,g',cm)) = add(g,g',add(f,f',cm))
  [cm4] add(f,f',cm) = add(f',f,cm)

  [cf1] conflict(f,f',empty-conflict-matrix) = false

  [cf2] and(not(eq(f,f')),
            or(and(eq(f,g), eq(f',g')),
               and(eq(f,g'), eq(f',g)))) = true
        ==>
        conflict(f,f',add(g,g',cm)) = true

  [cf3] and(not(eq(f,f')),
            or(and(eq(f,g), eq(f',g')),
               and(eq(f,g'), eq(f',g)))) = false
        ==>
        conflict(f,f',add(g,g',cm)) = conflict(f,f',cm)

  [co1] conflicts-of(f, empty-conflict-matrix) = empty-phase-name-set
  [co2] conflict(f,g,add(g,g',cm)) = true
        ==>
        conflicts-of(f,add(g,g',cm)) = add(g,conflicts-of(f,cm))
  [co3] conflict(f,g',add(g,g',cm)) = true
        ==>
        conflicts-of(f,add(g,g',cm)) = add(g',conflicts-of(f,cm))
  [co2] and(not(conflict(f,g,add(g,g',cm))),
            not(conflict(f,g',add(g,g',cm)))) = true
        ==>
        conflicts-of(f,add(g,g',cm)) = conflicts-of(f,cm)
end Conflict-Matrix
```

```
data module Signal-Status
begin
  exports
    begin
      sorts
        SIGNAL-STATUS
      functions
        yellow              : -> SIGNAL-STATUS
        yellow-intermittent : -> SIGNAL-STATUS
        white               : -> SIGNAL-STATUS
        white-intermittent  : -> SIGNAL-STATUS
        green               : -> SIGNAL-STATUS
        green-intermittent  : -> SIGNAL-STATUS
        red                 : -> SIGNAL-STATUS
        eq                  : SIGNAL-STATUS # SIGNAL-STATUS -> BOOLEAN
    end
  imports
    Numerical-Library
  functions
    I : SIGNAL-STATUS -> NATURAL
  variables
    st, st' :-> SIGNAL-STATUS
  equations
  [I1]  I(yellow)              = nat(^0)
  [I2]  I(yellow-intermittent) = nat(^1)
  [I3]  I(white)               = nat(^2)
  [I4]  I(white-intermittent)  = nat(^3)
  [I5]  I(green)               = nat(^4)
  [I6]  I(green-intermittent)  = nat(^5)
  [I7]  I(red)                 = nat(^6)
  [eq1] eq(st,st') = eq(I(st),I(st'))
  [eq2] eq(st,st') = eq(st',st)
end Signal-Status

data module Realization-Order
begin
  exports
    begin
      sorts
        REALIZATION-ORDER
      functions
        first-order  : -> REALIZATION-ORDER
        second-order : -> REALIZATION-ORDER
        third-order  : -> REALIZATION-ORDER
        initial-order: -> REALIZATION-ORDER
        eq           : REALIZATION-ORDER # REALIZATION-ORDER -> BOOLEAN
    end
  imports
    Numerical-Library
  functions
    I : REALIZATION-ORDER -> NATURAL
  variables
    ro, ro' :-> REALIZATION-ORDER
  equations
  [i0] I(initial-order) = nat(^0)
  [i1] I(first-order)   = nat(^1)
  [i2] I(second-order ) = nat(^2)
  [i3] I(third-order)   = nat(^3)
  [e1] eq(ro,ro') = eq(I(ro),I(ro'))
```

```
  [e2] eq(ro,ro') = eq(ro',ro)
end Realization-Order

data module Traffic-Participant
begin
  exports
    begin
      sorts
        TRAFFIC-PARTICIPANT
      functions
        pedestrian         : -> TRAFFIC-PARTICIPANT
        cyclist            : -> TRAFFIC-PARTICIPANT
        car                : -> TRAFFIC-PARTICIPANT
        eq                 : TRAFFIC-PARTICIPANT # TRAFFIC-PARTICIPANT
                             -> BOOLEAN
    end
  imports
    Numerical-Library
  functions
    I : TRAFFIC-PARTICIPANT -> NATURAL
  variables
    tp, tp' :-> TRAFFIC-PARTICIPANT
  equations
  [I1]  I(pedestrian) = nat(^0)
  [I2]  I(cyclist)    = nat(^1)
  [I3]  I(car)        = nat(^2)
  [eq1] eq(tp,tp') = eq(I(tp),I(tp'))
  [eq2] eq(tp,tp') = eq(tp',tp)
end Traffic-Participant

data module Traffic-Participant-and-Detection-Device-Operation
begin
  exports
    begin
      functions
        detected-for : DETECTION-DEVICE -> TRAFFIC-PARTICIPANT
    end
  imports
    Detection-Device, Traffic-Participant
  variables
    n :-> NATURAL
  equations
  [d1] detected-for(push-button-pedestrian(n)) = pedestrian
  [d2] detected-for(push-button-cyclist(n)) = cyclist
  [d3] detected-for(df-car(n)) = car
end Traffic-Participant-and-Detection-Device-Operation

data module Weight-Element
begin
  exports
    begin
      sorts
        WEIGHT-ELEM
      functions
        we : TRAFFIC-PARTICIPANT # NATURAL # NATURAL # NATURAL
             -> WEIGHT-ELEM
        eq : WEIGHT-ELEM # WEIGHT-ELEM -> BOOLEAN
    end
  imports
```

```
    Traffic-Participant
  variables
    tp, tp' : -> TRAFFIC-PARTICIPANT
    wn, wn' : -> NATURAL
    wf, wf' : -> NATURAL
    sn, sn' : -> NATURAL
    we, we' : -> WEIGHT-ELEM
  equations
  [eq1] eq(we(tp,wn,wf,sn), we(tp',wn',wf',sn'))
        = and(eq(tp, tp'), and(eq(wn, wn'), and(eq(wf, wf'), eq(sn, sn'))))
  [eq2] eq(we,we') = eq(we',we)
end Weight-Element

data module Weight-Element-Set
begin
  exports
    begin
      functions
        weight                    : WEIGHT-ELEM-SET  -> NATURAL
        traffic-waiting           : WEIGHT-ELEM-SET  -> BOOLEAN
        total-number-tp-counted   : WEIGHT-ELEM-SET  -> NATURAL
    end
  imports
    Set {Item bound by
           [ ITEM -> WEIGHT-ELEM, eq -> eq]
         to Weight-Element
         renamed by
           [ SET -> WEIGHT-ELEM-SET,
             empty -> empty-weight-elem-set]
        }
  variables
    tp                  : -> TRAFFIC-PARTICIPANT
    number,weight,total : -> NATURAL
    ws,rest-ws          : -> WEIGHT-ELEM-SET
  equations
  [w1]  weight(empty-weight-elem-set) = nat(^0)
  [w2]  weight(add(we(tp, number, weight, total), rest-ws))
        = total + weight(rest-ws)

  [t1]  traffic-waiting(ws) = gt(weight(ws),nat(^0))
  [tn1] total-number-tp-counted(empty-weight-elem-set) = nat(^0)
  [tn1] total-number-tp-counted(add(we(tp,number,weight,total),rest-ws))
        = number + total-number-tp-counted(rest-ws)
end Weight-Element-Set

data module Phase
begin
  exports
    begin
      sorts
        PHASE
      functions
        phase             : PHASE-NAME        #
                            REALIZATION-ORDER #
                            SIGNAL-STATUS     #
                            WEIGHT-ELEM-SET   #
                            NATURAL           #
                            NATURAL       -> PHASE
        name              : PHASE         -> PHASE-NAME
```

```
      realization-order : PHASE         -> REALIZATION-ORDER
      signal-status     : PHASE         -> SIGNAL-STATUS
      weight-elem-set   : PHASE         -> WEIGHT-ELEM-SET
      abs-time          : PHASE         -> NATURAL
      rest-time         : PHASE         -> NATURAL
      traffic-waiting   : PHASE         -> BOOLEAN
      eq                : PHASE # PHASE -> BOOLEAN
    end
  imports
    Phase-Name, Numerical-Library, Signal-Status, Weight-Element-Set,
    Realization-Order
  variables
    fn, fn' :-> PHASE-NAME
    ro, ro' :-> REALIZATION-ORDER
    ss, ss' :-> SIGNAL-STATUS
    ws, ws' :-> WEIGHT-ELEM-SET
    abs-time, abs-time', rest-time, rest-time' :-> NATURAL
    p, p'   :-> PHASE
  equations
  [n1]  name(phase(fn,ro, ss, ws, abs-time, rest-time)) = fn

  [r1]  realization-order(phase(fn, ro, ss, ws, abs-time, rest-time)) = ro

  [s1]  signal-status(phase(fn,ro, ss, ws, abs-time, rest-time)) = ss

  [w1]  weight-elem-set(phase(fn,ro, ss, ws, abs-time, rest-time)) = ws

  [a1]  abs-time(phase(fn,ro, ss, ws, abs-time, rest-time)) = abs-time

  [r1]  rest-time(phase(fn,ro, ss, ws, abs-time, rest-time)) = rest-time

  [tw1] traffic-waiting(phase(fn,ro,ss,ws,abs-time,rest-time))
        = gt(weight(ws), nat(^0))
  [eq1] eq(phase(fn,ro,ss,ws,abs-time,rest-time),
           phase(fn',ro',ss',ws',abs-time',rest-time'))
        = and(eq(fn,fn'),
              and(eq(ro,ro'),
                  and(eq(ss,ss'),
                      and(eq(ws,ws'),
                          and(eq(abs-time,abs-time'),
                              eq(rest-time,rest-time'))))))
  [eq2] eq(p,p') = eq(p',p)
end Phase

data module Phase-Set
begin
  exports
    begin
      functions
        names                     : PHASE-SET        -> PHASE-NAME-SET
        select-red                : PHASE-SET        -> PHASE-SET
        select-green              : PHASE-SET        -> PHASE-SET
        select-rest-time-zero     : PHASE-SET        -> PHASE-SET
        select-not-cleared        : PHASE-SET        -> PHASE-SET
        select-traffic-waiting    : PHASE-SET        -> PHASE-SET
        select-no-traffic-waiting : PHASE-SET        -> PHASE-SET
    end
  imports
    Phase-Name-Set,
```

```
  Set {Item bound by
         [ITEM -> PHASE, eq -> eq]
       to Phase
       renamed by
         [SET -> PHASE-SET, empty -> empty-phase-set]
      }
functions
  select-signal-status     : SIGNAL-STATUS # PHASE-SET     -> PHASE-SET
  select-realization-order : REALIZATION-ORDER # PHASE-SET -> PHASE-SET
variables
  f          :-> PHASE
  fs,rest-fs :-> PHASE-SET
  ss         :-> SIGNAL-STATUS
  ro         :-> REALIZATION-ORDER
equations
[n1]  names(empty-phase-set) = empty-phase-name-set
[n2]  names(add(f, rest-fs)) = add(name(f), names(rest-fs))

[s01] select-red(fs) = select-signal-status(red,fs)
[s02] select-green(fs) = select-signal-status(green,fs)

[s03] select-signal-status(ss,empty-phase-set) = empty-phase-set
[s04] eq(signal-status(f),ss) = true
      ==>
      select-signal-status(ss,add(f, rest-fs))
      = add(f, select-signal-status(ss,rest-fs))
[s05] eq(signal-status(f),ss) = false
      ==>
      select-signal-status(ss,add(f, rest-fs))
      = select-signal-status(ss,rest-fs)

[s06] select-rest-time-zero(empty-phase-set) = empty-phase-set
[s07] eq(rest-time(f), nat(^0)) = true
      ==>
      select-rest-time-zero(add(f, rest-fs))
      = add(f, select-rest-time-zero(rest-fs))
[s08] eq(rest-time(f), nat(^0)) = false
      ==>
      select-rest-time-zero(add(f, rest-fs))
      = select-rest-time-zero(rest-fs)

[s09] select-not-cleared(empty-phase-set) = empty-phase-set
[s10] gt(rest-time(f), nat(^0)) = true
      ==>
      select-not-cleared(add(f, rest-fs))
      = add(f, select-not-cleared(rest-fs))
[s11] gt(rest-time(f), nat(^0)) = false
      ==>
      select-not-cleared(add(f, rest-fs))
      = select-not-cleared(rest-fs)

[s12] select-traffic-waiting(empty-phase-set) = empty-phase-set
[s13] traffic-waiting(weight-elem-set(f)) = true
      ==>
      select-traffic-waiting(add(f, rest-fs))
      = add(f, select-traffic-waiting(rest-fs))
[s14] traffic-waiting(weight-elem-set(f)) = false
      ==>
      select-traffic-waiting(add(f, rest-fs))
```

```
        = select-traffic-waiting(rest-fs)

  [s15] select-no-traffic-waiting(empty-phase-set) = empty-phase-set
  [s16] traffic-waiting(weight-elem-set(f)) = false
        ==>
        select-no-traffic-waiting(add(f, rest-fs))
        = add(f, select-no-traffic-waiting(rest-fs))
  [s17] traffic-waiting(weight-elem-set(f)) = true
        ==>
        select-no-traffic-waiting(add(f, rest-fs))
        = select-no-traffic-waiting(rest-fs)
end Phase-Set

data module Account-Weighed-Priorities-Operation
begin
  exports
    begin
      functions
        account-weighed-priorities : PHASE-SET -> PHASE-SET
    end
  imports
    Phase-Set
  functions
    account-weighed-priorities : PHASE -> PHASE
    account-weighed-priorities : WEIGHT-ELEM-SET -> WEIGHT-ELEM-SET
  variables
    f                              :-> PHASE
    fs,fs',rest-fs                 :-> PHASE-SET
    fn                             :-> PHASE-NAME
    ro                             :-> REALIZATION-ORDER
    ss                             :-> SIGNAL-STATUS
    ws,rest-ws                     :-> WEIGHT-ELEM-SET
    abs-time,rest-time             :-> NATURAL
    number-tp,weight-factor,total :-> NATURAL
    tp                             :-> TRAFFIC-PARTICIPANT
  equations
  [w1] account-weighed-priorities(empty-phase-set)
       = empty-phase-set
  [w2] account-weighed-priorities(add(f, rest-fs))
       = add(account-weighed-priorities(f),
             account-weighed-priorities(rest-fs))

  [w3] account-weighed-priorities(phase(fn,ro,ss,ws,abs-time,rest-time))
       = phase(fn,ro,ss,account-weighed-priorities(ws),abs-time,rest-time)

  [w4] account-weighed-priorities(empty-weight-elem-set)
       = empty-weight-elem-set
  [w5] account-weighed-priorities(
         add(we(tp,number-tp,weight-factor,total),rest-ws))
       = add(we(tp,number-tp,weight-factor,total + (number-tp * weight-factor)),
             account-weighed-priorities(rest-ws))
end Account-Weighed-Priorities-Operation

data module Account-For-Time-Operation
begin
  parameters Actual-Situation
    begin
      functions
        duration-traffic-regulation-cycle : -> NATURAL
```

```
      clearance-duration                : PHASE-NAME -> NATURAL
      yellow-duration                   : PHASE-NAME -> NATURAL
  end Actual-Situation
exports
  begin
    functions
      account-for-time : PHASE-SET -> PHASE-SET
  end
imports
  Phase-Set
functions
  account-for-time : PHASE -> PHASE
  reset-weight-set : WEIGHT-ELEM-SET -> WEIGHT-ELEM-SET
variables
  abs-time, rest-time,
  number-tp,
  weight-factor, total :-> NATURAL
  f                    :-> PHASE
  fs, fs', rest-fs     :-> PHASE-SET
  fn                   :-> PHASE-NAME
  ro                   :-> REALIZATION-ORDER
  ss                   :-> SIGNAL-STATUS
  ws, rest-ws          :-> WEIGHT-ELEM-SET
  tp                   :-> TRAFFIC-PARTICIPANT
equations
[at1]  account-for-time(empty-phase-set) = empty-phase-set
[at2]  account-for-time(add(f,rest-fs))
       = add(account-for-time(f),account-for-time(rest-fs))

[at3]  eq(rest-time,nat(^0)) = true ==>
       account-for-time(phase(fn,ro,red,ws,abs-time,rest-time))
       = phase(fn,ro,red,ws,abs-time+duration-traffic-regulation-cycle,
               rest-time)
[at4]  gte(rest-time - duration-traffic-regulation-cycle, nat(^0)) = true
       ==>
       account-for-time(phase(fn,ro,red,ws,abs-time,rest-time))
       = phase(fn,ro,red,ws,abs-time + duration-traffic-regulation-cycle,
                rest-time  -  duration-traffic-regulation-cycle)

[at5]  eq(rest-time - duration-traffic-regulation-cycle, nat(^0)) = true
       ==>
       account-for-time(phase(fn,ro,green,ws,abs-time,rest-time))
       = phase(fn,ro,yellow,reset-weight-set(ws), nat(^0),
                 yellow-duration(fn))

[at6]  reset-weight-set(empty-weight-elem-set) = empty-weight-elem-set
[at7]  reset-weight-set(add(we(tp,number-tp,weight-factor,total),rest-ws))
       = add(we(tp,nat-zero,weight-factor,nat-zero),
            reset-weight-set(rest-ws))

[at8]  gt(rest-time - duration-traffic-regulation-cycle, nat(^0)) = true
       ==>
       account-for-time(phase(fn,ro,green,ws,abs-time,rest-time))
       = phase(fn,ro,green,ws,
               abs-time + duration-traffic-regulation-cycle,
               rest-time - duration-traffic-regulation-cycle)

[at9]  eq(rest-time - duration-traffic-regulation-cycle, nat(^0)) = true
       ==>
```

```
        account-for-time(phase(fn,ro,yellow,ws,abs-time,rest-time))
        = phase(fn,ro,red,ws,nat(^0), clearance-duration(fn))
  [at10] gt(rest-time - duration-traffic-regulation-cycle, nat(^0)) = true
        ==>
        account-for-time(phase(fn,ro,yellow,ws,abs-time,rest-time))
        = phase(fn,ro,yellow,ws,
                abs-time + duration-traffic-regulation-cycle,
                rest-time  -  duration-traffic-regulation-cycle)
end Account-For-Time-Operation

data module Realization-Toolkit
begin
  parameters Actual-Situation
    begin
      functions
        minimal-green-duration : PHASE-NAME              -> NATURAL
        clearance-duration     : PHASE-NAME              -> NATURAL
        yellow-duration        : PHASE-NAME              -> NATURAL
        higher-priority        : PHASE-NAME # PHASE-NAME -> BOOLEAN
        conflict-matrix        :                         -> CONFLICT-MATRIX
    end Actual-Situation
  exports
    begin
      functions
        estimate-green-duration : PHASE                  -> NATURAL
        highest-priority        : PHASE # PHASE-SET      -> PHASE
        conflict                : PHASE # PHASE-SET      -> BOOLEAN
        conflicts-in            : PHASE-NAME # PHASE-SET -> PHASE-SET
        spare-time              : PHASE-SET # PHASE-SET  -> NATURAL
        duration-to-clearance   : PHASE-SET              -> NATURAL-SET
    end
  imports
    Conflict-Matrix, Phase-Set, Natural-Set
  functions
    duration-to-clearance   : PHASE                      -> NATURAL
    retrieve                : PHASE-NAME-SET # PHASE-SET -> PHASE-SET
    realization-delay       : PHASE # PHASE-SET # PHASE-SET
                                                         -> NATURAL-SET
    realization-delay       : PHASE # PHASE-SET          -> NATURAL-SET
    spare-time              : NATURAL-SET                -> NATURAL
    time-to-wait            : NATURAL-SET                -> NATURAL
  variables
    fn, fn'                 :-> PHASE-NAME
    ro, ro'                 :-> REALIZATION-ORDER
    ss, ss'                 :-> SIGNAL-STATUS
    ws, ws', rest-ws        :-> WEIGHT-ELEM-SET
    abs-time, rest-time,
    abs-time', rest-time',n :-> NATURAL
    ns                      :-> NATURAL-SET
    f, f'                   :-> PHASE
    fs, fs', rest-fs,
    green-phases            :-> PHASE-SET
    fns                     :-> PHASE-NAME-SET
  equations
  [eg1] estimate-green-duration(f)
        = max(minimal-green-duration(name(f)),
              total-number-tp-counted(weight-elem-set(f)))
```

```
[d01] duration-to-clearance(phase(fn,ro,green,ws,abs-time,rest-time))
      = rest-time + yellow-duration(fn) + clearance-duration(fn)
[d02] duration-to-clearance(phase(fn,ro,yellow,ws,abs-time,rest-time))
      = rest-time + clearance-duration(fn)
[d03] duration-to-clearance(phase(fn,ro,red,ws,abs-time,rest-time))
      = rest-time
[d04] duration-to-clearance(empty-phase-set) = empty-natural-set
[d05] duration-to-clearance(add(f,rest-fs)) =
        add(duration-to-clearance(f),duration-to-clearance(rest-fs))

[hp1] highest-priority(f,empty-phase-set) = f
[hp2] or( gt(weight(weight-elem-set(f)), weight(weight-elem-set(f'))),
          and(eq(weight(weight-elem-set(f)),weight(weight-elem-set(f'))),
               higher-priority(name(f), name(f')))) = true
      ==>
      highest-priority(f, add(f', rest-fs)) = highest-priority(f,rest-fs)

[hp3] or( gt(weight(weight-elem-set(f)), weight(weight-elem-set(f'))),
           and(eq(weight(weight-elem-set(f)),weight(weight-elem-set(f'))),
               higher-priority(name(f), name(f')))) = false
      ==>
      highest-priority(f, add(f', rest-fs)) = highest-priority(f',rest-fs)

[rt1] retrieve(empty-phase-name-set,fs) = empty-phase-set
[rt2] retrieve(fns,empty-phase-set) = empty-phase-set
[rt3] is-in(name(f),fns) = true
      ==>
      retrieve(fns,add(f,rest-fs)) = add(f,retrieve(fns,rest-fs))
[rt4] is-in(name(f),fns) = false
      ==>
      retrieve(fns,add(f,rest-fs)) = retrieve(fns,rest-fs)

[co1] conflict(f,empty-phase-set) = false
[co2] conflict(f,add(f',rest-fs))
      = or(conflict(name(f),name(f'),conflict-matrix),
           conflict(f,rest-fs))

[co3] conflicts-in(fn,fs)
      = retrieve(conflicts-of(fn, conflict-matrix), fs)

[st1] spare-time(empty-phase-set,green-phases) = nat(^0)
[st1] spare-time(add(f,fs),green-phases)
      = spare-time(realization-delay(f,fs,green-phases))
[st2] spare-time(empty-natural-set) = nat(^0)
[st3] spare-time(add(n,ns)) = min(n,ns)

[rd1] realization-delay(f,empty-phase-set,fs')
      = add(time-to-wait(realization-delay(f,fs')),empty-natural-set)

[rd2] realization-delay(f,add(f',rest-fs),fs')
      = add(time-to-wait(realization-delay(f,fs')),
            realization-delay(f',rest-fs,fs'))

[rd3] realization-delay(f,empty-phase-set) = empty-natural-set
[rd4] conflict(name(f),name(f'),conflict-matrix) = true
      ==>
      realization-delay(f,add(f',rest-fs))
      = add(duration-to-clearance(f'),realization-delay(f,rest-fs))
```

```
  [rd5] conflict(name(f),name(f'),conflict-matrix) = false
        ==>
        realization-delay(f,add(f',rest-fs)) = realization-delay(f,rest-fs)

  [tw1] time-to-wait(empty-natural-set) = nat(^0)
  [tw2] time-to-wait(add(n,ns)) = max(n,ns)
end Realization-Toolkit


data module Realization-First-Order
begin
  exports
    begin
      functions
        realization-first-order   : PHASE-SET                    -> PHASE-SET
    end
  imports
    Realization-Toolkit
  functions
    realization-first-order-1 : PHASE-SET # PHASE-SET # PHASE-SET
                                                              -> PHASE-SET
    realization-first-order-2 : PHASE-SET # PHASE # PHASE-SET # PHASE-SET
                                                              -> PHASE-SET
    turn-green-first-order    : PHASE                         -> PHASE
    direct-conflict           : PHASE-SET # PHASE # PHASE-SET -> BOOLEAN
  variables
    fn                       :-> PHASE-NAME
    ro                       :-> REALIZATION-ORDER
    ws                       :-> WEIGHT-ELEM-SET
    abs-time, rest-time      :-> NATURAL
    f                        :-> PHASE
    fs,tw-zero-red,not-used,
    rest-tw-zero-red         :-> PHASE-SET
  equations
  [fo1] realization-first-order(fs)
        = realization-first-order-1(
            diff(fs,select-traffic-waiting(
                      select-rest-time-zero(
                        select-red(fs)))),
            select-traffic-waiting(
              select-rest-time-zero(
                select-red(fs))),
            empty-phase-set)

  [fo2] realization-first-order-1(fs,empty-phase-set,not-used)
        = union(fs,not-used)

  [fo3] realization-first-order-1(fs,add(f,rest-tw-zero-red),not-used)
        = realization-first-order-2(fs,
                                    highest-priority(f,rest-tw-zero-red),
                                    rem(highest-priority(f,rest-tw-zero-red),
                                        add(f,rest-tw-zero-red)),
                                    not-used)

  [fo4] direct-conflict(fs,f,not-used) = true
        ==>
        realization-first-order-2(fs,f,tw-zero-red,not-used)
        = realization-first-order-1(fs, tw-zero-red, add(f,not-used))
```

```
  [fo5] direct-conflict(fs,f,not-used) = false
        ==>
        realization-first-order-2(fs,f,tw-zero-red,not-used)
        = realization-first-order-1(
            add(turn-green-first-order(f),fs), tw-zero-red, not-used)

  [fo6] direct-conflict(fs,f,not-used)
        = or(conflict(f,not-used),
             conflict(f,diff(fs,select-no-traffic-waiting(
                                select-rest-time-zero(
                                  select-red(fs)))))
             )

  [tg1] turn-green-first-order(phase(fn,ro,red,ws,abs-time,rest-time))
        = phase(fn,first-order,green,ws,nat-zero,
            estimate-green-duration(phase(fn,ro,red,ws,abs-time,rest-time)))
end Realization-First-Order

data module Realization-Second-Order
begin
  exports
    begin
      functions
        realization-second-order  : PHASE-SET                    -> PHASE-SET
    end
  imports
    Realization-Toolkit
  functions
    realization-second-order-1: PHASE-SET # PHASE-SET # PHASE-SET -> PHASE-SET
    realization-second-order-2: PHASE-SET # PHASE #
                                PHASE-SET # PHASE-SET             -> PHASE-SET
    realization-second-order-3: PHASE-SET # PHASE #
                                PHASE-SET # PHASE-SET # NATURAL   -> PHASE-SET
    direct-conflict           : PHASE # PHASE-SET                 -> BOOLEAN
    turn-green-second-order   : PHASE # NATURAL                   -> PHASE
  variables
    f                             :-> PHASE
    fs,fs', tw-zero-red, not-used,
    rest-tw-zero-red              :-> PHASE-SET
    max-time, max-green-duration  :-> NATURAL
  equations
  [so1]  realization-second-order(fs)
         = realization-second-order-1(diff(fs,select-traffic-waiting(
                                             select-rest-time-zero(
                                               select-red(fs)))),
                                      select-traffic-waiting(
                                        select-rest-time-zero(
                                          select-red(fs))),
                                      empty-phase-set)

  [so2]  realization-second-order-1(fs,empty-phase-set,not-used)
         =
         union(fs,not-used)

  [so3]  realization-second-order-1(fs,add(f,rest-tw-zero-red),not-used)
         = realization-second-order-2(fs,highest-priority(f, rest-tw-zero-red),
                                      rem(highest-priority(f, rest-tw-zero-red),
                                          add(f, rest-tw-zero-red)),
                                      not-used)
```

```
  [so4]  direct-conflict(f,fs) = true
         ==>
         realization-second-order-2(fs,f,tw-zero-red,not-used)
         = realization-second-order-1(fs,tw-zero-red,add(f,not-used))

  [so5]  direct-conflict(f,fs) = false
         ==>
         realization-second-order-2(fs,f,tw-zero-red,not-used)
         = realization-second-order-3(fs,f,tw-zero-red,not-used,
                                      spare-time(
                                        conflicts-in(name(f),not-used),
                                          select-not-cleared(fs)))

  [so6]  gte(max-time,
             minimal-green-duration(name(f))
             + yellow-duration(name(f))
             + clearance-duration(name(f))) = true
         ==>
         realization-second-order-3(fs,f,tw-zero-red,not-used,max-time)
         = realization-second-order-1(
             add(turn-green-second-order(f,
                   max-time - yellow-duration(name(f))
                            - clearance-duration(name(f))),
                 fs),
             tw-zero-red, not-used)

  [so7]  gte(max-time,
             minimal-green-duration(name(f))
             + yellow-duration(name(f))
             + clearance-duration(name(f))) = false
         ==>
         realization-second-order-3(fs,f,tw-zero-red,not-used,max-time)
         = realization-second-order-1(fs,tw-zero-red,add(f,not-used))

  [so8]  direct-conflict(f,fs)
         = conflict(f,diff(fs,select-no-traffic-waiting(
                                select-rest-time-zero(
                                  select-red(fs)))))

  [so9]  gte(max-green-duration, minimal-green-duration(name(f))) = true
         ==>
         turn-green-second-order(f,max-green-duration)
         = phase(name(f), second-order, green, weight-elem-set(f), nat(^0),
                 min(max-green-duration, estimate-green-duration(f)))

  [so10] gte(max-green-duration, minimal-green-duration(name(f))) = false
         ==>
         turn-green-second-order(f,max-green-duration) = f
end Realization-Second-Order

data module Realization-Third-Order
begin
  exports
    begin
      functions
        realization-third-order   : PHASE-SET          -> PHASE-SET
    end
  imports
    Realization-Toolkit
```

```
functions
  realization-third-order-1 : PHASE-SET #
                              PHASE-SET # PHASE-SET   -> PHASE-SET
  realization-third-order-2 : PHASE-SET # PHASE #
                              PHASE-SET # PHASE-SET   -> PHASE-SET
  realization-third-order-3 : PHASE-SET # PHASE #
                              PHASE-SET # PHASE-SET #
                              PHASE-SET               -> PHASE-SET
  realization-third-order-4 : PHASE-SET # PHASE #
                              PHASE-SET # PHASE-SET #
                              NATURAL                 -> PHASE-SET
  direct-conflict           : PHASE # PHASE-SET       -> BOOLEAN
  turn-green-third-order    : PHASE # NATURAL         -> PHASE
  time-left                 : NATURAL-SET             -> NATURAL
variables
  fn, fn'                           :-> PHASE-NAME
  f                                 :-> PHASE
  fs, ntw-zero-red, not-used,
  overrun, rest-ntw-zero-red        :-> PHASE-SET
  max-time, max-green-duration, n   :-> NATURAL
  ns                                :-> NATURAL-SET
equations
[to1a] eq(select-green(fs),empty-phase-set) = true
       ==>
       realization-third-order(fs) = fs

[to1b] eq(select-green(fs),empty-phase-set) = false
       ==>
       realization-third-order(fs)
       = realization-third-order-1(diff(fs,select-no-traffic-waiting(
                                           select-rest-time-zero(
                                             select-red(fs)))),
                                   select-no-traffic-waiting(
                                     select-rest-time-zero(
                                       select-red(fs))),
                                   empty-phase-set)

[to2]  realization-third-order-1(fs,empty-phase-set,not-used)
       = union(fs,not-used)

[to3]  realization-third-order-1(fs,add(f,rest-ntw-zero-red),not-used)
       = realization-third-order-2(fs,
                                   highest-priority(f, rest-ntw-zero-red),
                                   rem(highest-priority(f, rest-ntw-zero-red),
                                       add(f, rest-ntw-zero-red)),
                                   not-used)

[to4]  direct-conflict(f,fs) = true
       ==>
       realization-third-order-2(fs,f,ntw-zero-red,not-used)
       = realization-third-order-1(fs,ntw-zero-red,add(f,not-used))

[to5]  direct-conflict(f,fs) = false
       ==>
       realization-third-order-2(fs,f,ntw-zero-red,not-used)
       = realization-third-order-3(fs,f,ntw-zero-red,not-used,
                                   conflicts-in(name(f),select-traffic-waiting
                                                  select-rest-time-zero
                                                    select-red(fs)))))
```

```
  [to5a] eq(overrun, empty-phase-set) = true
         ==>
         realization-third-order-3(fs,f,ntw-zero-red,not-used,overrun)
         = realization-third-order-4(fs,f,ntw-zero-red,not-used,
                                     time-left(duration-to-clearance(
                                                 select-green(fs))))

  [to5b] eq(overrun, empty-phase-set) = false
         ==>
         realization-third-order-3(fs,f,ntw-zero-red,not-used,overrun)
         = realization-third-order-4(fs,f,ntw-zero-red,not-used,
                                     spare-time(
                                       overrun,
                                       select-not-cleared(fs)))

  [to6]  gte(max-time,
             minimal-green-duration(name(f))
             + yellow-duration(name(f))
             + clearance-duration(name(f))) = true
         ==>
         realization-third-order-4(fs,f,ntw-zero-red,not-used,max-time)
         =
         realization-third-order-1(
           add(
             turn-green-third-order(f,
               max-time - yellow-duration(name(f))
                        - clearance-duration(name(f))),
             fs),
           ntw-zero-red, not-used)

  [to7]  gte(max-time,
             minimal-green-duration(name(f))
             + yellow-duration(name(f))
             + clearance-duration(name(f))) = false
         ==>
         realization-third-order-4(fs,f,ntw-zero-red,not-used,max-time)
         = realization-third-order-1(fs,ntw-zero-red,add(f,not-used))

  [to8]  direct-conflict(f,fs)
         = conflict(f,diff(fs,select-traffic-waiting(
                                select-rest-time-zero(
                                  select-red(fs)))))

  [to9]  gte(max-green-duration, minimal-green-duration(name(f))) = true
         ==>
         turn-green-third-order(f,max-green-duration)
         = phase(name(f), third-order, green, weight-elem-set(f), nat(^0),
                 min(max-green-duration, estimate-green-duration(f)))

  [to10] gte(max-green-duration, minimal-green-duration(name(f))) = false
         ==>
         turn-green-third-order(f,max-green-duration) = f

  [to11] time-left(empty-natural-set) = nat(^0)
  [t012] time-left(add(n,ns)) = min(n,ns)
end Realization-Third-Order
```

```
data module Pair
begin
  parameters Item1
    begin
      sorts
        ITEM1
      functions
        eq   : ITEM1 # ITEM1 -> BOOLEAN
    end Item1,
    Item2
    begin
      sorts
        ITEM2
      functions
        eq   : ITEM2 # ITEM2 -> BOOLEAN
    end Item2
  exports
    begin
      sorts
        PAIR
      functions
        pair      : ITEM1 # ITEM2 -> PAIR
        proj1     : PAIR          -> ITEM1
        proj2     : PAIR          -> ITEM2
        eq        : PAIR # PAIR   -> BOOLEAN
    end
  imports Booleans
  variables
    i1,i1' :-> ITEM1
    i2,i2' :-> ITEM2
  equations
  [P1] proj1(pair(i1,i2)) = i1
  [P2] proj2(pair(i1,i2)) = i2
  [E]  eq(pair(i1,i2),pair(i1',i2')) = and(eq(i1,i1'), eq(i2,i2'))
end Pair

data module Detection
begin
  imports
    Pair {Item1 bound by
            [ITEM1 -> DETECTION-DEVICE, eq -> eq]
          to Detection-Device
          Item2 bound by
            [ITEM2 -> BOOLEAN, eq -> eq]
          to Booleans
          renamed by
            [PAIR -> DETECTION, pair -> detection]
         }
end Detection

data module Detection-Set
begin
  imports
    Set {Item bound by
           [ITEM -> DETECTION, eq -> eq]
         to Detection
         renamed by
           [SET -> DETECTION-SET, empty -> empty-detection-set] }
end Detection-Set
```

```
data module Account-For-Detection-Operation
begin
  parameters Actual-Situation
    begin
      functions
        detected-for-phase : DETECTION-DEVICE -> PHASE-NAME
    end Actual-Situation
  exports
    begin
      functions
        account-for-detection : DETECTION-SET # PHASE-SET -> PHASE-SET
    end
  imports
    Traffic-Participant-and-Detection-Device-Operation,
    Detection-Set,Phase-Set
  functions
    account-for-detection : DETECTION-SET # PHASE -> PHASE
    account-for-detection : DETECTION-DEVICE #
                            WEIGHT-ELEM-SET -> WEIGHT-ELEM-SET
    account-for-detection : DETECTION-DEVICE # NATURAL -> NATURAL
  variables
    f               :-> PHASE
    detector        :-> DETECTION-DEVICE
    fs,fs',rest-fs  :-> PHASE-SET
    ds, rest-ds     :-> DETECTION-SET
    fn              :-> PHASE-NAME
    ro              :-> REALIZATION-ORDER
    ss              :-> SIGNAL-STATUS
    ws, rest-ws     :-> WEIGHT-ELEM-SET
    t1,t2,n1,n2,n3  :-> NATURAL
    tp              :-> TRAFFIC-PARTICIPANT
  equations
  [ad1] account-for-detection(ds,empty-phase-set) = empty-phase-set
  [ad2] account-for-detection(ds,add(f,rest-fs))
        = add(account-for-detection(ds,f),account-for-detection(ds,rest-fs))

  [ad3] account-for-detection(empty-detection-set,f) = f
  [ad4] account-for-detection(add(detection(detector,false),rest-ds),f)
        = account-for-detection(rest-ds,f)
  [ad5] eq(detected-for-phase(detector),fn) = true ==>
        account-for-detection(add(detection(detector,true),rest-ds),
          phase(fn,ro,ss,ws,t1,t2))
        = account-for-detection(rest-ds,
            phase(fn,ro,ss,account-for-detection(detector,ws),t1,t2))
  [ad6] eq(detected-for-phase(detector),fn) = false ==>
        account-for-detection(add(detection(detector,true),rest-ds),
          phase(fn,ro,ss,ws,t1,t2))
        = account-for-detection(rest-ds,phase(fn,ro,ss,ws,t1,t2))

  [ad7] account-for-detection(detector,empty-weight-elem-set)
        = empty-weight-elem-set
  [ad8] eq(detected-for(detector),tp) = true ==>
        account-for-detection(detector,add(we(tp,n1,n2,n3),rest-ws))
        = add(we(tp,account-for-detection(detector,n1),n2,n3),rest-ws)
  [ad9] eq(detected-for(detector),tp) = false ==>
        account-for-detection(detector,add(we(tp,n1,n2,n3),rest-ws))
        = account-for-detection(detector,rest-ws)
```

```
  [ad10] account-for-detection(push-button-pedestrian(n1),n2) = nat(^1)
  [ad11] account-for-detection(push-button-cyclist(n1),n2)    = nat(^1)
  [ad12] account-for-detection(df-car(n1),n2)                 = n2 + nat(^1)
  [ad13] account-for-detection(df-public-transport(n1),n2)    = n2 + nat(^1)
  [ad14] account-for-detection(df-assistance-service(n1),n2)  = n2 + nat(^1)
end Account-For-Detection-Operation

data module Operation
begin
  imports
    Account-For-Detection-Operation
      {Actual-Situation bound by [detected-for-phase -> detected-for-phase]
                        to Actual-Situation-Dataset
      },
    Account-Weighed-Priorities-Operation,
    Account-For-Time-Operation
      {Actual-Situation bound by [duration-traffic-regulation-cycle ->
                                  duration-traffic-regulation-cycle,
                                  clearance-duration -> clearance-duration,
                                  yellow-duration -> yellow-duration]
                        to Actual-Situation-Dataset
      },
    Realization-First-Order
      {Actual-Situation
       bound by [minimal-green-duration -> minimal-green-duration,
                 clearance-duration -> clearance-duration,
                 yellow-duration -> yellow-duration,
                 higher-priority -> higher-priority,
                 conflict-matrix -> conflict-matrix]
       to Actual-Situation-Dataset
      },
    Realization-Second-Order
      {Actual-Situation
       bound by [minimal-green-duration -> minimal-green-duration,
                 clearance-duration -> clearance-duration,
                 yellow-duration -> yellow-duration,
                 higher-priority -> higher-priority,
                 conflict-matrix -> conflict-matrix]
       to Actual-Situation-Dataset
      },
    Realization-Third-Order
      {Actual-Situation
       bound by [minimal-green-duration -> minimal-green-duration,
                 clearance-duration -> clearance-duration,
                 yellow-duration -> yellow-duration,
                 higher-priority -> higher-priority,
                 conflict-matrix -> conflict-matrix]
       to Actual-Situation-Dataset
      }
end Operation

process module Control-Process
begin
  parameters Actual-Situation
    begin
      functions
        init-status        : -> PHASE-SET
        poll-first         : -> DETECTION-DEVICE
        poll-last          : -> DETECTION-DEVICE
```

```
      next-2-poll        : DETECTION-DEVICE -> DETECTION-DEVICE
  end Actual-Situation
exports
  begin
    atoms
      detect             : DETECTION-DEVICE # BOOLEAN
      change-signal      : PHASE
    processes
      CONTROL-PROCESS
  end
imports
  Operation
processes
  Control-Process        : PHASE-SET
  Poll-Detection-Devices : PHASE-SET
  Poll-Detection-Devices : DETECTION-DEVICE # DETECTION-SET # PHASE-SET
  Normal-Procedure       : PHASE-SET
  Change-Signals         : PHASE-SET
variables
  status, new-status     :-> PHASE-SET
  f                      :-> PHASE
  detector               :-> DETECTION-DEVICE
  ds                     :-> DETECTION-SET
definitions
  CONTROL-PROCESS
  = Control-Process(init-status)

  Control-Process(status)
  = Poll-Detection-Devices(status)

  Poll-Detection-Devices(status)
  = Poll-Detection-Devices(poll-first,empty-detection-set,status)

  Poll-Detection-Devices(detector,ds,status)
  = [  eq(detector,poll-last) = false]
    -> sum(b in BOOLEAN,
             detect(detector,b) .
             Poll-Detection-Devices(next-to-poll(detector),
                                    add(detection(detector,b),ds),status)
          )
    +
    [  eq(detector,poll-last) = true]
    -> sum(b in BOOLEAN,
             detect(detector,b) .
             Normal-Procedure(
              realization-third-order(
               realization-second-order(
                realization-first-order(
                 account-for-time(
                  account-weighed-priorities(
                   account-for-detection(
                    add(detection(detector,b),ds),status)))))))
          )

  Normal-Procedure(new-status)
  = Change-Signals(new-status) . Control-Process(new-status)

  Change-Signals(empty-phase-set)
  = skip
```

```
    Change-Signals(add(f, status))
    = change-signal(f) . Change-Signals(status)
end Control-Process

process module Traffic-Regulation-System
begin
  exports
    begin
      processes
        TRS
    end
  imports
    Control-Process
      { Actual-Situation
          bound by [init-status -> initial-status,
                    poll-first -> first-to-poll,
                    poll-last -> last-to-poll,
                    next-2-poll -> next-to-poll]
          to Actual-Situation-Dataset
      }
  definitions
    TRS = CONTROL-PROCESS
end Traffic-Regulation-System
```

# C Example of an actual situation data set

This data set could serve for the intersection in the example in section 3.

```
data module Actual-Situation-Dataset
begin
  exports
    begin
      functions
        conflict-matrix               :                   -> CONFLICT-MATRIX
        initial-status                :                   -> PHASE-SET
        detected-for-phase            : DETECTION-DEVICE -> PHASE-NAME
        clearance-duration            : PHASE-NAME       -> NATURAL
        yellow-duration               : PHASE-NAME       -> NATURAL
        minimal-green-duration        : PHASE-NAME       -> NATURAL
        first-to-poll                 :                   -> DETECTION-DEVICE
        last-to-poll                  :                   -> DETECTION-DEVICE
        next-to-poll                  : DETECTION-DEVICE -> DETECTION-DEVICE
        duration-traffic-regulation-cycle
                                      :                   -> NATURAL
        higher-priority               : PHASE-NAME #
                                        PHASE-NAME       -> BOOLEAN
    end
  imports
    Conflict-Matrix, Detection-Device, Phase-Set
  functions
    prior : PHASE-NAME -> NATURAL
  variables
    n       :-> NATURAL
    fn, fn' :-> PHASE-NAME
  equations
  [c1] conflict-matrix
       = add(01, 05, add(01, 09,
         add(02, 05, add(02, 06, add(02, 09, add(02, 10, add(02, 11, add(02, 12,
         add(03, 05, add(03, 06, add(03, 07, add(03, 08, add(03, 11, add(03, 12,
         add(04, 08, add(04, 12,
         add(05, 08, add(05, 09, add(05, 12,
         add(06, 08, add(06, 09, add(06, 10, add(06, 11,
         add(07, 11,
         add(08, 11, add(08, 12,
         add(09, 11, add(09, 12,
         empty-conflict-matrix))))))))))))))))))))))))))))

  [i1] initial-status
       = add(phase(01,initial-order,red,
                   add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                   nat(^0),nat(^0)),
         add(phase(02,initial-order,red,
                   add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                   nat(^0),nat(^0)),
         add(phase(03,initial-order,red,
                   add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                   nat(^0),nat(^0)),
         add(phase(04,initial-order,red,
                   add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                   nat(^0),nat(^0)),
```

```
        add(phase(05,initial-order,red,
                  add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                  nat(^0),nat(^0)),
        add(phase(06,initial-order,red,
                  add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                  nat(^0),nat(^0)),
        add(phase(07,initial-order,red,
                  add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                  nat(^0),nat(^0)),
        add(phase(08,initial-order,red,
                  add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                  nat(^0),nat(^0)),
        add(phase(09,initial-order,red,
                  add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                  nat(^0),nat(^0)),
        add(phase(10,initial-order,red,
                  add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                  nat(^0),nat(^0)),
        add(phase(11,initial-order,red,
                  add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                  nat(^0),nat(^0)),
        add(phase(12,initial-order,red,
                  add(we(car,nat(^0),nat(^1),nat(^0)), empty-weight-elem-set),
                  nat(^0),nat(^0)),
            empty-phase-set))))))))))))

[p1]   prior(01) = nat(^1)
[p2]   prior(02) = nat(^2)
[p3]   prior(03) = nat(^3)
[p4]   prior(04) = nat(^4)
[p5]   prior(05) = nat(^5)
[p6]   prior(06) = nat(^6)
[p6]   prior(07) = nat(^7)
[p7]   prior(08) = nat(^8)
[p9]   prior(09) = nat(^9)
[p10]  prior(10) = nat(^1^0)
[p11]  prior(11) = nat(^1^1)
[p12]  prior(12) = nat(^1^2)
[p13]  higher-priority(fn, fn') = gt(prior(fn), prior(fn'))

[d1]   detected-for-phase(df-car(nat(^1))) = 01
[d2]   detected-for-phase(df-car(nat(^2))) = 02
[d3]   detected-for-phase(df-car(nat(^3))) = 03
[d4]   detected-for-phase(df-car(nat(^4))) = 04
[d5]   detected-for-phase(df-car(nat(^5))) = 05
[d6]   detected-for-phase(df-car(nat(^6))) = 06
[d7]   detected-for-phase(df-car(nat(^7))) = 07
[d8]   detected-for-phase(df-car(nat(^8))) = 08
[d9]   detected-for-phase(df-car(nat(^9))) = 09
[d10]  detected-for-phase(df-car(nat(^1^0))) = 10
[d11]  detected-for-phase(df-car(nat(^1^1))) = 11
[d12]  detected-for-phase(df-car(nat(^1^2))) = 12

[cd1]  clearance-duration(01) = nat(^1)
[cd2]  clearance-duration(02) = nat(^1)
[cd3]  clearance-duration(03) = nat(^1)
[cd4]  clearance-duration(04) = nat(^1)
[cd5]  clearance-duration(05) = nat(^1)
[cd6]  clearance-duration(06) = nat(^1)
```

```
  [cd7]  clearance-duration(07) = nat(^1)
  [cd8]  clearance-duration(08) = nat(^1)
  [cd9]  clearance-duration(09) = nat(^1)
  [cd10] clearance-duration(10) = nat(^1)
  [cd11] clearance-duration(11) = nat(^1)
  [cd12] clearance-duration(12) = nat(^1)

  [y1]   yellow-duration(01) = nat(^1)
  [y2]   yellow-duration(02) = nat(^1)
  [y3]   yellow-duration(03) = nat(^1)
  [y4]   yellow-duration(04) = nat(^1)
  [y5]   yellow-duration(05) = nat(^1)
  [y6]   yellow-duration(06) = nat(^1)
  [y7]   yellow-duration(07) = nat(^1)
  [y8]   yellow-duration(08) = nat(^1)
  [y9]   yellow-duration(09) = nat(^1)
  [y10]  yellow-duration(10) = nat(^1)
  [y11]  yellow-duration(11) = nat(^1)
  [y12]  yellow-duration(12) = nat(^1)

  [m1]   minimal-green-duration(01) = nat(^2)
  [m2]   minimal-green-duration(02) = nat(^2)
  [m3]   minimal-green-duration(03) = nat(^2)
  [m4]   minimal-green-duration(04) = nat(^2)
  [m5]   minimal-green-duration(05) = nat(^2)
  [m6]   minimal-green-duration(06) = nat(^2)
  [m7]   minimal-green-duration(07) = nat(^2)
  [m8]   minimal-green-duration(08) = nat(^2)
  [m9]   minimal-green-duration(09) = nat(^2)
  [m10]  minimal-green-duration(10) = nat(^2)
  [m11]  minimal-green-duration(11) = nat(^2)
  [m12]  minimal-green-duration(12) = nat(^2)

  [fp1]  first-to-poll = df-car(nat(^1))
  [lp1]  last-to-poll  = df-car(nat(^1^2))
  [np1]  next-to-poll(df-car(n)) =
         df-car(mod(n,nat(^1^2)) + nat(^1))

  [dr1]  duration-traffic-regulation-cycle = nat(^1)
end Actual-Situation-Dataset
```