A Real Time μCRL Specification of a System for Traffic Regulation at Signalized Intersections

Michiel J. Koens
koens@fwi.uva.nl 8940614

Liang H. Oei
lhoei@fwi.uva.nl 9017291

Abstract

This paper takes up the challenge advanced in Report P9313 of the Programming Research Group of the University of Amsterdam [VW93]. It provides an alternative way of specifying an existing traffic regulation system at signalized intersections of the firm **Nederland Haarlem**[1]. The specification proposed in this paper features the major differences with the previous attempt, which are threefold:

1. it is process oriented rather than data oriented, i.e. each phase is modeled as a process that runs in parallel with the others. Furthermore there is one central control process that decides which phases are realized (turn green).
2. it uses the $\rho\mu$CRL (Real Time Micro Common Represention Language), a Real Time extension of μCRL , instead of PSF (Process Specification Formalism);
3. it features Prefix Integrated Real Time Algebra of Communicating Processes (ACP).

Key words & phrases: algebraic specification, real time process algebra, $\rho\mu$CRL , traffic regulation.

1 Introduction

This paper is the final term of the course Real Time Process Algebra at the University of Amsterdam and brings a real time specification of a Traffic Regulation System. As specification language we chose μCRL as described in [GP90]. We did not use the time extended version $r\mu$CRL as found in [Fok92], because it uses absolute time instead of relative time, and it has a notion of integration different from the kind we were familiar with. μCRL features data, but is untimed, so we added prefixed integration and relative time and named this extension $\rho\mu$CRL . The theory on which these additions are based can be found in [Klu93], which is based on [BB91].

We would like to stress here that as far as we know our attempt to specify such a large domain in $\rho\mu$CRL has no precedence and brings about all the obvious experimental hazards one can think of. Our specification of a traffic regulation system is based upon the work of Bas van Vlijmen and Arjan van Waveren [VW93]. They in turn have based their work on Nederland Haarlem documents written by Jan Kroone [Kro90].

In this paper we first will give a short description of the problem to be specified. After a brief intermezzo of some theoretical backgrounds we then give a thorough explanation of the specification. The next chapter describes the major differences with the specification as found in [VW93]. Finally the specification itself is appended.

2 Acknowledgments

We would like to express our gratitude to the following persons : Jan Bergstra for his provocative manner of teaching and his interest for reading our draft, Steven Klusener for his dedication to his

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

assistantship and his innumerable tips and hints, Bas van Vlijmen for being the in between with Nederland Haarlem and Willem Jan Fokkink for his draft of rμCRL syntax in ASF-SDF, which has motivated us in using Real Time μCRL . We would like to thank Nederland Haarlem especially for giving us the opportunity to present and communicate our ideas.

3 Dynamic Traffic Regulation System at Signalized Intersections

Within this paper we will maintain the jargon proposed by Nederland Haarlem as to maintain downward compatibility with the origins of this specification. In our strive for conciseness we have decided to describe the problem by a list of definitions, which is essentially a summary of the exposition given by [VW93] and indirectly also of [Kro90]:

3.1 Basic Traffic Jargon

3.1.1 Objects

Streams are subsections of the influx of traffic participants into the intersection scene from one particular road branch, e.g., one queuing section for cars wanting to turn left of a T-intersection would be one indivisible stream. To each stream a constant weight factor is attributed.

Signalized intersections are traffic intersections where every stream is regulated with a signal; typically a traffic light.

Phases are collections of streams synchronized by effectively the same signal. This signal could be split into a special car, bicycle or pedestrian signal, but as the behavior of these signals are completely synchronous, these various streams would make one phase. The meaning of *phase* is therefore idiomatic for Nederland Haarlem and should not be confused with the colloquial semantics of changing states.
The following constants are attributed to each phase:

- a minimal green time,
- a clearance duration,
- a yellow time,
- and an absolute priority, to be able to order phases when their weighed waiting times are the same.

In our specification each phase has been modeled as a process that accounts for its own weighed waiting time, which depends linearly on the number of traffic participants waiting and their weight factors. (see below: **Weighed priorities accounting**).

Detection devices are associated with each stream. In our specification each stream has one detection device. Nederland Haarlem specifies detectors for pedestrians to be able to detect one pedestrian only. They also carry detection devices that are sensitive to special traffic like ambulances. Since the internal behavior of detection devices is not our goal of study, we have modeled detectors as atomic actions without communications. The detection of a traffic participant then becomes just an event.

3.1.2 Objectives

Realization of a phase means that its signal turned green. A phase that has been cleared (see definition below) and has traffic waiting is requesting for realization.

Clearance duration of a phase is the time constant associated with the phase in which it is expected to have 'dispatched' its realized traffic participants and has effectively forced all incoming traffic to a standstill. Popularly speaking: the phase has been red long enough after being green and yellow so as not to endanger further development on the critical area of the intersection.

3.2 Realization paraphernalia

3.2.1 The four order regulation system

The four order regulation system is a proprietary dynamic scheduling algorithm developed by Nederland Haarlem, a company that is leading the market of traffic regulation systems. The sophistication of this algorithm is expressed by not only achieving the primary goal of a traffic regulation system, namely: the absence of collisions, but also in the optimalization of throughput without frustrating the sense of fairness associated with dynamic configuration of waiting traffic participants with respect to time.

The algorithm of the four order regulation system resides in the control process. The control process loops over a time constant that in real life situations amounts to one second. The control process has been modeled by a single process within our specification.

3.2.2 The conflict matrix

A conflict matrix, the matrix of pairs of conflicting phases (phases that physically cannot be realized at the same time — since they would simply collide — form pairs), is associated with the control process as its main datastructure. [2]

3.2.3 Dynamic priority accounting

Like in the previous study the dynamic priority of each phase is calculated each second. It is determined by the weighed waiting time of the phases and their absolute priorities in case some waiting times are equal.

The weighed waiting time is calculated each accounting cycle (in our specific case: one tenth of a second). In every cycle it is increased with the product of the number of traffic participants waiting and their associated weight factor.

3.2.4 First order realization

A phase will be realized first order if:

1. it is requesting realization,
2. it has higher priority than any other conflicting phase requesting realization,
3. and is not in conflict with non-cleared phases (read: moving traffic).

3.2.5 Second order realization

A phase will be realized second order if:

1. it is requesting realization, but cannot be realized first order,
2. and it does not frustrate any waiting phases of higher priority,
3. and is not in conflict with non-cleared phases.

[2] Note that the specified conflict matrix here is like in [VW93] a simplification of the real conflict matrix which contains a lot more data.

Intuitively: a phase can be realized second order, because the phases with higher priority it could block, are already blocked by another phase. A phase realized in second order is just traffic that is 'covered' by a phase that will get a higher order realization or a phase that is non-cleared.

3.2.6 Third order realization

A phase will be realized third order if:

1. it is cleared and it has no traffic waiting,
2. and it can be realized without frustrating a cleared phase with a higher priority that has no traffic waiting,
3. and it can be realized without frustrating a phase with a request for realization pending,
4. and it is not in conflict with any of the non-cleared phases.

Within our specification the third order realization algorithm is immersed in the second order realization function: phases with no traffic waiting get the lowest priorities, so they are treated last automatically in our specification, without frustrating others.

3.2.7 Fourth order realization

Fourth order realizations are realizations granted to vehicles owning special privileges like: fire patrol, police, ambulance, public transportation and the like. This feature does not figurate in our specification, but we feel justified in mentioning it since the nomenclature of the algorithm would otherwise be very puzzling.

3.2.8 Green time accounting

The green time granted to a realized phase into first order is taken as the maximum of the minimal greentime and an estimate of the requested greentime. As a simplification the latter is equated to the number of traffic participants waiting.

Since a second order realization is not allowed to frustrate other waiting phases, it must be cleared again at the moment the first of the conflicting waiting phases with a higher priority is no longer frustrated by another realized phase.

4 Some Theoretical Backgrounds

4.1 Syntax: Actions, Relative Time, Prefixed Integration

Time stamped actions take no time and can be performed immediately after each other. $a[t]$ denotes an action a to be performed t time steps after the previous action. [3]

An integral preceding a time stamped action is used to give the possibility to perform an action at *some* time given by a condition, typically a time interval.

A silent step (or internal step) denoted by the greek letter τ, is the abstraction from internal behaviour. Communications are renamed to τ's.

[3] In this document actions are implicitly *urgent* actions as described in [Klu93], e.g. $a[0]$ means that a must be executed *immediately* by which we allow consecutive actions at the same point in time.

The square brackets explicitly determines the relative time setting. An alternative would be absolute time denoted by $a(t)$.

4.2 Examples of Process Algebra Semantics

- $a[2] \cdot b[1]$ means: after waiting two time steps perform action a and after waiting another time step perform action b; the $\cdot$ is the sequential operator.
- $a[2] \cdot b[0]$ means: perform after two time steps a then immediately b.
- $a[2] + b[4]$ means: perform either a after two time steps or b after four time steps. The $+$ is the alternative operator; the choice is nondeterministic. After waiting two time steps this process has been evolved into $a[0] + b[2]$, idling forth determines a choice for the b; by idling another time step $a[0] + b[2]$ evolves to $b[1]$. Note that $a[0] + b[2]$ can also execute the action a immediately.
- $P \| Q$ means: perform P in parallel with Q.
- $\int_{v>2}(a[v] \cdot b[v+1])$ means: perform action a after waiting some period $v > 2$ and then perform b after waiting another period $v+1$.
- $a(p : Pid)[t]$ means: action a parametrized with the data type Pid. $a(1)[t]$ would be in this context an action a instantiated by the process-id '1'.
- $\sum_{p \in Pid} Pcleared(p)$ means: the generalized summation over the alternative compositions over the set of process identifiers, it rewrites to: $Pcleared(1) + Pcleared(2) + ... + Pcleared(12)$ in case $Pid = 1, \ldots, 12$.
- $P(w : Time) = \int_{v \in [0,w\rangle} detect[v] \cdot P(w - v) + \tau[w] \cdot P(cycletime)$ means: the process P can either detect traffic participants between now and time w and then call itself with the remaining part of w, or wait until w and perform the internal action τ after which it calls itself, where w is set to $cycletime$.

5 The Specification

5.1 The Process Equations

5.1.1 The Phase Processes

The P processes describe the behavior of a phase.

- $P^{cleared}_{red}$ describes the behavior of a phase when it is cleared and turned red.

$$P^{cleared}_{red}(p : Pid, wt : WeighedTimeTable, w : Time) =$$

1. Its first parameter gives its identification number p, its Pid.
2. The next parameter is a table wt, containing information about the number of traffic participants waiting at each of its stream, the total number of traffic participants waiting and a weighed waiting time.
3. The last parameter is w, denoting the time it has to wait to recalculate its weighed waiting time.

This process can perform four actions. The first three actions can be performed during an interval from immediately to the time given by w.

$$\int_{v \in [0,w\rangle} \sum_{s \in Sid} detect(s)[v] \cdot P^{cleared}_{red}(p, update(wt, s), w - v)$$

In this interval it can detect a traffic participant at any of its streams, after which it calls itself with an updated wt and the remaining part of w.

$$\int_{v\in[0,w\rangle} send_{wt}(p, weighedTime(wt), totalTp(wt))[v] \cdot P_{red}^{cleared}(p, wt, w-v)$$

It can also send the control process a tuple, holding the number of waiting traffic participants and the weighed waiting time. After this action the process calls itself as well.

$$\int_{v\in[0,w\rangle} \sum_{t\in Time} rec_{realization}(p,t)[v] \cdot set_{green}(p)[0] \cdot P_{green}(p,t)$$

The third possibility is to receive a realization message from the control process, holding the length of the period it has to be green. After receiving such a message the process performs a set_{green}, turning the lights green, and changes its behavior by calling P_{green}. P_{green} needs its Pid and the greentime, so these are given as parameters.

$$\tau[w] \cdot P_{red}^{cleared}(p, calculate_{wt}(p, wt), cycletime)$$

The last alternative of $P_{red}^{cleared}$ is to do nothing until the end of the aforementioned period (by performing a $\tau[w]$) and call itself again with a table wt with updated weighed waiting time and a new time ($cycletime$) to denote when to update the weighed waiting time again.

$$\begin{array}{l} P_{red}^{\neg cleared}(p : Pid, wt : WeighedTimeTable, w : Time) \\ = \\ \int_{v\in[0,w\rangle} \sum_{s\in Sid} detect(s)[v] \cdot \ P_{red}^{\neg cleared}(p, update(wt,s), w-v) \\ +\tau[w] \cdot P_{red}^{cleared}(p, calculate_{wt}(wt), cycletime) \end{array}$$

- $P_{red}^{\neg cleared}$ has a similar behavior as $P_{red}^{cleared}$, but it will never receive a message to turn green from the control process, nor will it be asked to send its waiting information. The w denotes the clearance duration. After this period the phase is cleared and $P_{red}^{cleared}$ is called.

$$P_{green}(p : Pid, t : Time) = set_{yellow}(p)[t] \cdot P_{yellow}(p)$$

- P_{green} is called immediately after the lights at a phase are turned green. The second parameter gives the time after which it must turn the lights to yellow and call P_{yellow}.

$$\begin{array}{rcl} P_{yellow}(p : Pid) & = & set_{red}(p)[yellowtime(p)] \\ & & \cdot\, P_{red}^{\neg cleared}(p, initstreams(p), clearance(p)) \end{array}$$

- P_{yellow} turns the lights from yellow to red again. The period after which this must happen is a constant given by the function $yellowtime$.

5.1.2 The Control Process

The CP processes describe the behavior of the control process.

- CP_{rec} (which abbreviates $CP_{receive}$) has three parameters.

  1. A set of cleared phases $\langle p, cs\rangle$ (first element is p) [4],
  2. a set of non-cleared phases ncs and
  3. a set ws of cleared phases which have sent their waiting information.

$$\begin{aligned} & CP_{rec}(\langle p, cs\rangle : PidSet, ncs : PendingSet, ws : WaitingSet) \\ = & \\ & \sum_{t \times tp \in Time \times Int} rec_{wt}(p, t, tp)[0] \cdot CP_{rec}(cs, ncs, \langle p, t, tp, ws\rangle) \end{aligned}$$

The first parameter gives the set of phases that may be realized. CP_{rec} asks with rec_{wt} a phase p in this set for its waiting information: its weighed waiting time (t) and the number of traffic participants waiting (tp).

The received information is stored with the Pid of the phase in ws and CP_{rec} calls itself with the updated ws and a cs where the processed phase is left out.

$$\begin{aligned} & CP_{rec}(\emptyset : PidSet, ncs : PendingSet, ws : WaitingSet) \\ = & \\ & CP_{send}(peel(ws), ncs, realizations(ncs, ws)) \end{aligned}$$

When the set of cleared phases is empty (all cleared phases have sent their waiting information), CP_{send} is called with the result of the function *realizations*. This function calculates the dynamic priority of the phases and then finds which phases can be realized for how long. The result of *realizations* is a list of pairs of a phase and the time that phase must be turned green. To be able to prevent conflicts it needs the set of phases that are non-cleared as an extra parameter. This set can be derived by *peel*ing the time part and the traffic participants part from ws.

- CP_{send} is called with a set of cleared phases, a set of non-cleared phases and the set of phases that have to be realized together with their greentime.

$$\begin{aligned} & CP_{send}(cs : PidSet, ncs : PendingSet, \langle p, t, ps\rangle : PendingSet) \\ = & \\ & send_{realization}(p, t)[0] \\ & \cdot CP_{send}(extract(p, cs), \langle p, t + yellowtime(p) + clearance(p), ncs\rangle, ps) \end{aligned}$$

CP_{send} takes a Pid p and a greentime from its third argument and sends the phase with Pid p the message to turn green for a period t.

It then calls itself with updated parameters: p is removed from the set of cleared phases cs and added to the set of non-cleared phases ncs and the processed phase is removed from the Pending-list denoting the set of phases that must set to green.

[4] A set is implemented as a list, which has a first element, to be able to walk through the set.

The time after which a phase is cleared is given by its greentime + its *yellowtime* + its *clearance* duration. This is the time that is stored with p in *ncs*.

$$
\begin{array}{l}
CP_{send}(cs : PidSet, ncs : PendingSet, \emptyset : PendingSet) \\
= \\
\tau[controlcycletime] \cdot CP_{rec}(newcs(cs, ncs), newncs(ncs), \emptyset)
\end{array}
$$

When all the phases that must turn green are informed (*ps* is empty) the control process waits a period *controlcycletime* and starts a new control cycle by calling CP_{rec} again. This is done with updated *ncs* and *cs*: the duration to clearance of every non-cleared phase as stored in *ncs* is decremented by *controlcycletime*.

When a duration to clearance is (less than or equal to) zero it means that this particular phase is cleared now and its Pid is moved from *ncs* to *cs*.

5.1.3 The Traffic Regulation System Process

Finally, process *RTRS* describes the overall process.

$$
\begin{array}{lcl}
RTRS & = & \tau_{\{comm_{realization}, comm_{wt}\}}(\\
 & & \partial_{\{send_{wt}, rec_{wt}, send_{realization}, rec_{realization}\}}(\\
 & & CP_{rec}(intersectionPids, \emptyset, \emptyset) \| \\
 & & \|_{p \in intersectionPids} \; (P_{red}^{cleared}(p, initstreams(p), cycletime))))
\end{array}
$$

It starts a *CP* merged with one *P* for every phase in the particular intersection. All these processes are initialized with values specified in the actual intersection part of the specification. By renaming isolated send and receive actions to δ (*deadlock*) we force these actions to be performed in pairs, resulting in a communication.

These resulting communications are internal and are renamed to τ (*silent step*). We note that a hierarchy emerges because `CP` determines the time to communicate with `P`, while `P` accepts communication using an interval.

5.1.4 The Process Equations put together

$$
\begin{array}{l}
P_{red}^{cleared}(p : Pid, wt : WeighedTimeTable, w : Time) \\
= \\
\int_{v \in [0,w\rangle} \sum_{s \in Sid} detect(s)[v] \cdot P_{red}^{cleared}(p, update(wt, s), w - v) \\
+ \int_{v \in [0,w\rangle} send_{wt}(p, weighedTime(wt), totalTp(wt))[v] \cdot P_{red}^{cleared}(p, wt, w - v) \\
+ \int_{v \in [0,w\rangle} \sum_{t \in Time} rec_{realization}(p, t)[v] \cdot set_{green}(p) \cdot P_{green}(p, t) \\
+ \tau[w] \cdot P_{red}^{cleared}(p, calculate_{wt}(p, wt), cycletime)
\end{array}
$$

$$
\begin{array}{l}
P_{red}^{\neg cleared}(p : Pid, wt : WeighedTimeTable, w : Time) \\
=
\end{array}
$$

$$\int_{v\in[0,w\rangle} \sum_{s\in Sid} detect(s)[v]$$
$$\cdot P_{red}^{\neg cleared}(p, update(wt,s), w-v)$$
$$+\tau[w] \cdot P_{red}^{cleared}(p, calculate_{wt}(wt), cycletime)$$

$$P_{green}(p : Pid, t : Time) \;=\; set_{yellow}(p)[t] \cdot P_{yellow}(p)$$

$$\begin{aligned} P_{yellow}(p : Pid) \;&=\; set_{red}(p)[yellowtime(p)] \\ &\quad \cdot P_{red}^{\neg cleared}(p, initstreams(p), clearance(p)) \end{aligned}$$

$$CP_{rec}(\emptyset : PidSet, ncs : PendingSet, ws : WaitingSet)$$
$$=$$
$$CP_{send}(peel(ws), ncs, realizations(ncs, ws))$$

$$CP_{rec}(\langle p, cs\rangle : PidSet, ncs : PendingSet, ws : WaitingSet)$$
$$= \sum_{t\times tp\in Time\times Int} rec_{wt}(p,t,tp)[0] \cdot \; CP_{rec}(cs, ncs, \langle p, t, tp, ws\rangle)$$

$$CP_{send}(cs : PidSet, ncs : PendingSet, \emptyset : PendingSet)$$
$$=$$
$$\tau[controlcycletime] \cdot CP_{rec}(newcs(cs, ncs), newncs(ncs), \emptyset)$$

$$CP_{send}(cs : PidSet, ncs : PendingSet, \langle p, t, ps\rangle : PendingSet)$$
$$=$$
$$\begin{aligned} &send_{realization}(p,t)[0] \cdot CP_{send}(extract(p, cs), \\ &\qquad \langle p, t + yellowtime(p) + clearance(p), ncs\rangle, ps) \end{aligned}$$

$$\begin{aligned} RTRS \;=\; &\tau_{\{comm_{realization}, comm_{wt}\}}(\\ &\partial_{\{send_{wt}, rec_{wt}, send_{realization}, rec_{realization}\}}(\\ &CP_{rec}(intersectionPids, \emptyset, \emptyset) \| \\ &\Big\|_{p\,\in\, intersectionPids} \; (P_{red}^{cleared}(p, initstreams(p), cycletime)))) \end{aligned}$$

5.2 Abstract Data Types and Operators

We will now explain the remaining part of the specification: the sorts and functions of the Abstract Datatypes

5.2.1 Generic Datatypes from Standard Libraries

As μCRL does not have "built-in" datatypes we have to provide specifications even for the simple datatypes as booleans and integers. The previous study of the Traffic Regulation System made use of a standard library of Abstract Data Type primitives: The PSF Library [Wam91]. Since μCRL does not allow us to import a library, we had to adapt these modules by hand.

- The sorts `Bool` and `Int` are self-explanatory. Integers, or numbers in general are especially handy for coding Object Id's. The sorts for instance `Pid` and `Sid` are used to identify phases and streams within phases. They are mapped to integers.

5.2.2 $\rho\mu$CRL Time Datatype

We specified sort `Time` as the Rational Numbers, because we were not able to specify Reals in a simple fashion. This means that our specification is not as 'Real' as the title might suggest. An action at time sqrt(2) cannot occur. On the other hand an action very near to sqrt(2) with any desired precision can be modeled.

The result of a calculation is not simplified, i.e. `plus(R(1,2),R(1,2))` $\rightarrow$ `R(2,2)` ($\frac{1}{2}+\frac{1}{2} \rightarrow \frac{2}{2}$), but this does not cause problems, as we have an equality function `eq` as well by which `eq(R(2,2), R(1,1))` reduces to true.

A sort `Bool` and a sort `Time` are essential in $\rho\mu$CRL and should be supplied with every specification [Fok92]. However, Jan Bergstra commented that this still leaves enough degrees of freedom.

5.2.3 Phase related datastructures and operators

- Sort `WeighedTimeTable` provides a table for all actual information a phase needs to keep. It holds the accounted weighed priority for the phase and its accumulated traffic participants for its associated set of streams. `calculateWeight` recalculates the weighed waiting time and `update` is used when a new traffic participant is detected and the list of streams must be updated.

5.2.4 Control Process related datastructures and operators

- Sort `ConflictMatrix` lets the user build a conflict matrix by adding conflicting-pairs to an `emptyConflictMatrix`. A `PidSet` is used to keep track of sets of `Pid`'s, for instance the set of cleared phases cs as used in the process equations.
- A `WaitingSet` is used to keep a set (ws in the equations) of phases together with their weighed waiting time and number of traffic participants.
- A 'sort'-function, using the insertion sort algorithm, is defined to tell `realizations` which phases in the list have the highest priority to be set to green first order.
- A `PendingSet` holds a set of (`Pid x Time`)-pairs. We have seen the `PendingSet`s ncs and ps in the process equations.
- Functions `newCs` and `newNcs` are used after every cycle of the control process to determine which phases are cleared and moved from `ncs` to `cs`.

5.2.5 The Realization Algorithm

- The functions `realizations`, `rIterate`, `r1` and `r2` are used to find which phases for how long should be turned green. They are the core of this specification.
- `rIterate` finds phases that are not in conflict with non-cleared phases. If such a phase is not in conflict with a non-realizable phase with higher priority (a phase in `notUsed`) it is realized first order by `r1`, otherwise it is realized second order by `r2`, if the time given is long enough.
- The available time is calculated by `spareTime`. This function first finds the phases (say P) in `notUsed` it can frustrate. Then it finds the time the first of these phases can be realized; it takes the time the first phase in `ncs` that frustrate phases in P is cleared. This minimum can be found, i.e. the set of phases is not empty, because the function `spareTime` is only called when a phase `p` conflicts with `notUsed` and every phase in `notUsed` has a conflict with one or more phases in `ncs`.

5.2.6 Actual Intersection Data

Finally, the data constants are given which determine the behavior of the example intersection to be specified, which in our case has been given the same values as in Appendix C of [VW93]. Unlike [VW93] we have not filled in an instance of a traffic situation complete with detected traffic participants for the simple reason that there are no tools yet available to simulate a $\rho\mu$CRL specification in Real Time.

6 A Comparison with the Previous Study

We started specifying using the prose of [VW93]. Their specification was only used for fine-tuning our specification.

The most important differences between the two specifications are described in this chapter.

6.1 Data versus Processes

A system with an inherent control process like the Traffic Regulation System can be naturally specified using the Algebra of Communicating Processes (see [BW90]). This writing considers phases as processes instead of records in a datastructure of a control process. This enhances the readability of the specification considerably.

The nestings in the original specification of the Control-Process disappeared because more functions are called in the loops of the process-definitions. To illustrate the difference we take the liberty to 'quote' a fragment of the previous study in which only one artificial Control-Process is defined which triggers the accounting function. Due the lack of other processes, no communication or parallelism is defined.

The greatest disadvantage of the previous study is that at the top level the *states* of the phases is hidden away into lower levels. We think phases of the Traffic Regulation System at the Intersection level are too important to be factored out at the top module by abstraction. The behavior of phases is the point of interest when we study the system on this intermediate level.

```
% fragment taken from Appendix B. from [VlijWa93] p. 63/64

process module Control-Process
begin
...
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) . ChangeSignals(status)
end Control-Process
```

The original specification carries phases with all information around in the determination of the phases to be realized. We have chosen to carry minimal information around, but this resulted in sorts `PidSet`, `WaitingSet` and `PendingSet` and the need for functions `weightFactor` and `cumulative` to instruct function `update`. Those different lists gave rise to different versions of much-used functions (`conflict`, `in`, `if`, etc.). An alternative would be to use one kind of list that can hold all information needed, and fill it with dummies when necessary.

6.2 PSF versus $\rho\mu$CRL

The advantages we tasted while using $\rho\mu$CRL are unsurprisingly the lack of overhead features that were eliminated from PSF at the birth of μCRL. Especially the absence of the import and export facility and a great deal of typical ASF-PSF syntactical sugaring contribute to the reduction of the number of lines in the specification. But the lack of these facilities makes it very questionable whether a *complete* specification, where all necessary details for the final implementation are included and the nestings of functions and sorts are deep, should be specified in $\rho\mu$CRL. [5]

The original specification is more explicit (meaning: things are more explicitly spelled out), which causes the specification to be larger and (maybe) less efficient. Our specification is more compact, despite the absence of pre-defined sorts and functions. Compare for instance the two methods to determine which phases to realize.

6.3 Timeless versus Real Time

The original specification of the Traffic Regulation System relies heavily on the notion of cycli. These are assumed to be discrete time steps that can be modeled in loop counters. That approach worked, but the use of time as in our specification has some advantages: the notion of time is explicit. This is especially important when a genuine time critical system is to be specified. Real Time μCRL is in one way more expressive than PSF. The use of integrals provides a continuous timing, which may be difficult to express by clock-simulating processes in PSF. Moreover, writing the processcycle as timestep implies that the timing of the system depends implicitly on the speed of the hardware.

[5] The modular structure of a PSF specification enables the specification of a hierarchy of separate modules, each with their local definitions. Unfortunately *Completeness* of the associated term rewriting system is not a modular property. A counter example given by Barendregt and Klop (1986) is reproduced in [BKM89]. For proof theoretical reasons modularity is not supported by μCRL.

The behavior of our specification is different, because the P-processes calculate their weighed waiting time ten times a second (the *cycletime* is $\frac{1}{10}$ time step) and are asked to send this information every second (the *controlcycletime* is 1 time step). We added this extra cycle to the specification, partly to show the expressiveness of $\rho\mu$CRL. It is mentioned on page 20 of [VW93] as a property of the real system by Nederland Haarlem.

Our proposed features for $\rho\mu$CRL, however, are used without taking too much into consideration what consequences they have on the operational semantics of μCRL. For instance, we have only looked at the operators we needed for this specific instance of a Traffic Regulation System. Even on this experimental level there are enough differences between our syntax definition and the one given by [Fok92], most notably our usage of prefixed integration and our decision to settle for relative time instead of absolute time. This should give rise for a comparison and is an area for further study.

7 Conclusion

Algebraic Specification forces one into an analytical thought process. A description of a solution of a particular problem in the form of an algebraic specification is therefore less error prone than one created with a not so rigorous formal method. This sounds like an open door but only by 'doing it' we have learned to 'feel' that way.

We have learned to appreciate Algebraic Specification as a Formal Document describing the data definition and process behavior of a system whereupon equational reasoning can be applied. We have experienced that it is unwise to optimize during the specification activity since this seriously endangers the readability of the specification and very easily introduces ambiguities and ironically enough also redundancies.

We have shown that Real Time Process Algebra embedded in μCRL is a considerable enrichment of the expressivity of μCRL. It would be a very tedious exercise indeed to perform this reverse engineering project the other way around, namely by creating a separate process to synchronize with a Timer process for each time stamped process in our specification. Not to mention the communication overhead.

The introduction of Asynchronous Communicating Processes in our distributed Traffic Regulation System specification adds another level of functional decomposition to the original specification. This does not mean, however, that our specification dictates a possible implementation to be executed in parallel. The specification as given in the appendix could be implemented as one monolithic block as well. We would argue that this is the virtue of abstraction.

References

[BB91] J.C.M Baeten & J.A. Bergstra, Real time process algebra. *Journal of Formal Aspects of Computing Science*, 3(2) : 142-188, 1991.

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

[BKM89] J.A. Bergstra, J.W. Klop & A. Middeldorp, *Termherschrijfsystemen*, PTT Research 1989, Kluwer Programmatuurkunde, in Dutch.

[Fok92] W.J. Fokkink, *A simple specification language combining processes, time and data.* Center for Mathematics and Computer Science, Technical Report CS-R9209, Department of Software Technology, Amsterdam 1992.

[GP90] J.F. Groote & A. Ponse, *The Syntax and Semantics of μCRL.* Center for Mathematics and Computer Science, Technical Report CS-R9076, Department of Software Technology, Amsterdam 1990.

[Kli93] P. Klint, A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176-201, 1993.

[Klu93] A.S. Klusener, *Models and axioms for a fragment of real time process algebra.* Ph.D thesis, Technical University of Eindhoven, December 1993.

[Kro90] J.J.M.T. Kroone, *UAP-26 Universele applicatieprogramma's voor de FR34 en FR80.* B.V. Nederland Haarlem Verkeerssystemen, 1990.

[Mau91] S. Mauw, *PSF a process specification formalism.* Ph.D thesis, University of Amsterdam 1991.

[Vel91] G.J. Veltink, *The PSF Toolkit.* Technical Report P9107, Programming Research Group, University of Amsterdam 1991.

[VW93] S.F.M. van Vlijmen & A. van Waveren, *Algebraic Specification of a System for Traffic Regulation at Signalized Intersections.* Technical Report P9313, Programming Research Group, University of Amsterdam June 1993.

[Wam91] J.J.van Wamel *A Library for PSF*, Technical Report P9301, Programming Research Group, University of Amsterdam, 1993.

A The Specification

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                             %
%      Real Time Micro Common Representation Language Specification           %
%     of a System for Traffic Regulation at Signalized Intersections          %
%                   by Michiel J. Koens & Liang H. Oei                        %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Begin Standard Sorts

sort Bool
func T, F        :             -> Bool
     or, and, xor: Bool # Bool -> Bool
     not         : Bool        -> Bool
     if : Bool # Bool # Bool   -> Bool
var  b, b0       : Bool
rew  or(T, b)  = T
     or(F, b)  = b
     and(T, b) = b
     and(F, b) = F
     xor(T, b) = not(b)
     xor(F, b) = b
     not(T)    = F
     not(F)    = T
     if(T, b, b0) = b
     if(F, b, b0) = b0

sort Nat % only used to specify Ints
func zero             :           -> Nat
     s                : Nat       -> Nat
     plus, sbtr, mlty : Nat # Nat -> Nat
     leq, eq          : Nat # Nat -> Bool
var  nat, nat0        : Nat
rew  plus(nat, zero)        = nat
     plus(nat, s(nat0))     = s(plus(nat, nat0))
     sbtr(nat, zero)        = nat
     sbtr(zero, s(nat0))    = zero
     sbtr(s(nat),s(nat0))   = sbtr(nat, nat0)
     mlty(nat, zero)        = zero
     mlty(nat, s(nat0))     = plus(mlty(nat, nat0), nat)
     leq(zero, nat)         = T
     leq(s(nat), s(nat0))   = leq(nat, nat0)
     leq(s(nat), zero)      = F
     eq(zero, zero)         = T
     eq(s(nat), zero)       = F
     eq(zero, s(nat))       = F
     eq(s(nat), s(nat0))    = eq(nat, nat0)

sort Sign % only used to specify Ints
func Pos, Neg    :             -> Sign
     eq          : Sign # Sign -> Bool
var  sign, sign0 : Sign
rew  eq(Pos, Pos) = T
     eq(Neg, Neg) = T
     eq(Pos, Neg) = F
     eq(Neg, Pos) = F
```

```
sort Int
func 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12: -> Int
     I                : Sign # Nat             -> Int % constructor
     plus, sbtr, mlty : Int # Int              -> Int
     if               : Bool # Int # Int       -> Int
     leq, eq          : Int # Int              -> Bool
     pos              : Int                    -> Bool
var  n, n0         : Int
     sign, sign0   : Sign
     nat, nat0     : Nat
rew  if(T, n, n0) = n
     if(F, n, n0) = n0
     plus(I(sign, nat), I(Neg, nat0))   = sbtr(I(sign, nat), I(Pos, nat0))
     plus(I(Neg, nat), I(Pos, nat0))    = sbtr(I(Pos, nat0), I(Pos, nat))
     plus(I(Pos, nat), I(Pos, nat0))    = I(Pos, plus(nat, nat0))
     sbtr(I(sign, nat), I(Neg, nat0))   = plus(I(sign, nat), I(Pos, nat0))
     sbtr(I(Neg, nat), I(Pos, nat0))    = I(Neg, plus(nat, nat0))
     sbtr(I(Pos, nat), I(Pos, nat0))    = if(leq(nat, nat0),
                                             I(Neg, sbtr(nat0, nat)),
                                             I(Pos, sbtr(nat, nat0)))
     mlty(I(sign, nat), I(sign0, nat0)) = if(eq(sign, sign0),
                                             I(Pos, mlty(nat, nat0)),
                                             I(Neg, mlty(nat, nat0)))

     eq(I(sign, nat), I(sign0, nat0))   =
       or(and(eq(sign, sign0), eq(nat, nat0)),
          and(eq(nat, zero), eq(nat0, zero)))

     leq(I(Pos, nat), I(Pos, nat0))     = leq(nat, nat0)
     leq(I(Neg, nat), I(Pos, nat0))     = T
     leq(I(Pos, nat), I(Neg, nat0))     = and(eq(nat, zero), eq(nat0, zero))
     leq(I(Neg, nat), I(Neg, nat0))     = leq(nat0, nat)

     pos(n)                             = leq(0, n)

      0 = I(Pos, zero)
      1 = I(Pos, s(zero))
      2 = I(Pos, s(s(zero)))
      3 = I(Pos, s(s(s(zero))))
      4 = I(Pos, s(s(s(s(zero)))))
      5 = I(Pos, s(s(s(s(s(zero))))))
      6 = I(Pos, s(s(s(s(s(s(zero)))))))
      7 = I(Pos, s(s(s(s(s(s(s(zero))))))))
      8 = I(Pos, s(s(s(s(s(s(s(s(zero)))))))))
      9 = I(Pos, s(s(s(s(s(s(s(s(s(zero))))))))))
     10 = I(Pos, s(s(s(s(s(s(s(s(s(s(zero)))))))))))
     11 = I(Pos, s(s(s(s(s(s(s(s(s(s(s(zero))))))))))))
     12 = I(Pos, s(s(s(s(s(s(s(s(s(s(s(s(zero)))))))))))))

sort Time % defined as the Rational numbers
func R                          : Int # Int             -> Time % constructor
     plus, sbtr, min, max       : Time # Time           -> Time
     leq, lt, eq                : Time # Time           -> Bool
     if                         : Bool # Time # Time    -> Time
     time                       : Int                   -> Time % typecast
var  t, t0                      : Time
     n, n0, n1, n2              : Int
```

```
     sign                         : Sign
     nat                          : Nat
rew  plus(R(n, n0), R(n1, n2)) =
       R(plus(mlty(n, n2), mlty(n1, n0)), mlty(n0, n2))
     sbtr(R(n, n0), R(n1, n2)) =
       R(sbtr(mlty(n, n2), mlty(n1, n0)), mlty(n0, n2))
     leq(R(n, n0), R(n1, n2))  = if(xor(pos(n0), pos(n2)),
                                    leq(mlty(n0, n1), mlty(n, n2)),
                                    leq(mlty(n, n2), mlty(n0, n1)))
     eq(t, t0)                    = and(leq(t, t0), leq(t0, t))
     lt(t, t0)                    = and(leq(t, t0), not(leq(t0, t)))
     min(t, t0)                   = if(leq(t, t0), t, t0)
     max(t, t0)                   = if(leq(t, t0), t0, t)
     if(T, t, t0)                 = t
     if(F, t, t0)                 = t0

% End Standard Sorts

% Begin Global Data Structures Definitions

sort Sid % Stream Id
func S             : Int       -> Sid % constructor
     int           : Sid       -> Int % typecast
     eq            : Sid # Sid -> Bool
var  s, s0         : Sid
     n             : Int
rew  int(S(n))     = n
     eq(s, s0)     = eq(int(s), int(s0))

sort Pid % Phase Id
func P                                       : Int       -> Pid % constructor
     int                                     : Pid       -> Int % typecast
     eq                                      : Pid # Pid -> Bool
var  p, p0                                   : Pid
     n                                       : Int
rew  int(P(n)) = n
     eq(p, p0) = eq(int(p), int(p0))

sort ConflictMatrix
func add        : Pid # Pid # ConflictMatrix             -> ConflictMatrix
     emptyConflictMatrix, conflictMatrix:                -> ConflictMatrix
     conflict   : Pid # Pid                              -> Bool
     in         : Pid # Pid # ConflictMatrix             -> Bool
     if : Bool # ConflictMatrix # ConflictMatrix -> ConflictMatrix
var  m, m0      : ConflictMatrix
     p, p0, p1, p2 : Pid
rew  add(p, p0, add(p1, p2, m)) =
       if(or(and(eq(p, p1), eq(p0, p2)),
             and(eq(p, p2), eq(p0, p1))),
               add(p1, p2, m),
               add(p1, p2, add(p, p0, m)))

     if(T, m, m0) = m
     if(F, m, m0) = m0

     conflict(p, p0) = if(leq(int(p), int(p0)),
                         in(p, p0, conflictMatrix),
                         in(p0, p, conflictMatrix))
```

```
     in(p, p0, emptyConflictMatrix) = F
     in(p, p0, add(p1, p2, m))     =
       if(eq(p, p0),
         or(and(eq(p, p1), eq(p0, p2)), in(p, p0, m)),
 or(and(eq(p0, p1), eq(p, p2)), in(p0, p, m)))

% End Global Data Structures Definitions

% Begin Actual Intersection Data Set

func weightFactor                                : Pid # Sid -> Int
     priority                                    : Pid       -> Int
     cumulative                                  : Pid # Sid -> Bool
     cycleTime, controlCycleTime                 :           -> Time
     yellowTime, minGreenTime, clearanceDuration : Pid       -> Time
     actualIntersectionPidSet                    :           -> PidSet
     initStreams                                 : Pid       -> WeighedTimeTable
var  n : Int
     p : Pid
     s : Sid
rew                cycleTime  = R(1, 10)
           controlCycleTime   = R(1, 1)

               priority(P(n)) = n
                yellowTime(p) = R(1, 1)
              minGreenTime(p) = R(2, 1)
         clearanceDuration(p) = R(1, 1)

          weightFactor(p, s)  = 1
             cumulative(p, s) = T    % COUNT number of detects

     conflictMatrix =
       add(P(1),  P(5), add(P(1), P(9),
       add(P(2),  P(5), add(P(2), P(6),  add(P(2), P(9),  add(P(2), P(10),
                                         add(P(2), P(11), add(P(2), P(12),
       add(P(3),  P(5), add(P(3), P(6),  add(P(3), P(7),  add(P(3), P(8),
                                         add(P(3), P(11), add(P(3), P(12),
       add(P(4),  P(8), add(P(4), P(12),
       add(P(5),  P(8), add(P(5), P(9),  add(P(5), P(12),
       add(P(6),  P(8), add(P(6), P(9),  add(P(6), P(10), add(P(6), P(11),
       add(P(7), P(11),
       add(P(8), P(11), add(P(8), P(12),
       add(P(9), P(11), add(P(9), P(12),
       emptyConflictMatrix))))))))))))))))))))))))))

     actualIntersectionPidSet =
       add(P(1), add(P(2), add(P(3), add(P(4), add(P(5), add(P(6), add(P(7),
       add(P(8), add(P(9), add(P(10), add(P(11), add(P(12),
       emptyPidSet))))))))))))

     initStreams(P(n)) = Wt(R(0,1), 0, add(S(1), 0, emptyStreamDataSet))
       % n not used since as a simplification we assume every phase to
       % have one stream as in [VlijWa93].

% End Actual Intersection Data Set

% Begin Dynamic Accounting Functions Definitions
```

```
sort PidSet
func add        : Pid # PidSet                    -> PidSet
     emptyPidSet:                                 -> PidSet
     extract, selectConflict : Pid # PidSet       -> PidSet
     conflict   : Pid # PidSet                    -> Bool
     in         : Pid # PidSet                    -> Bool
     if         : Bool # PidSet # PidSet          -> PidSet
var  notUsed, cs, pset, pidset, pidset0 : PidSet
     p, p0                              : Pid
rew  add(p, add(p0, pidset)) = if(eq(p, p0),
                               add(p0, pidset),
                               add(p0, add(p, pidset)))

     extract(p, emptyPidSet)     = emptyPidSet  % error
     extract(p, add(p0, pidset)) =
       if(eq(p, p0), pidset, add(p0, extract(p, pidset)))

     conflict(p, emptyPidSet)   = F
     conflict(p, add(p0, pset)) = or(conflict(p0, p), conflict(p, pset))

     selectConflict(p, emptyPidSet)   = emptyPidSet
     selectConflict(p, add(p0, pidset)) =
       if(conflict(p, p0),
          add(p0, selectConflict(p, pidset)),
          selectConflict(p, pidset))

     in(p, emptyPidSet)   = F
     in(p, add(p0, pidset)) = or(eq(p, p0), in(p, pidset))

     if(T, pidset, pidset0) = pidset
     if(F, pidset, pidset0) = pidset0

sort StreamDataSet WeighedTimeTable
func add                    : Sid # Int # StreamDataSet    -> StreamDataSet
     emptyStreamDataSet     :                              -> StreamDataSet
     Wt                     : Time # Int # StreamDataSet   -> WeighedTimeTable
     weighedTime            : WeighedTimeTable             -> Time
     totalTp                : WeighedTimeTable             -> Int
     calculateWeight        : Pid # WeighedTimeTable       -> WeighedTimeTable
     update                 : Pid # WeighedTimeTable # Sid -> WeighedTimeTable
     update                 : Pid # StreamDataSet # Sid    -> StreamDataSet
     cwIterate: Time # StreamDataSet # StreamDataSet       -> WeighedTimeTable
     cwIterate: Pid # Time # Int # StreamDataSet # StreamDataSet
                                                           -> WeighedTimeTable
     if: Bool # WeighedTimeTable # WeighedTimeTable        -> WeighedTimeTable
     if: Bool # StreamDataSet # StreamDataSet              -> StreamDataSet
var  wt, wt0                : WeighedTimeTable
     sset, sset0            : StreamDataSet
     n, n0                  : Int
     s, s0                  : Sid
     p, p0                  : Pid
     t, t0                  : Time
rew  add(s, n, add(s0, n0, sset)) = if(eq(s, s0),
                                   add(s0, n0, sset),
                                   add(s0, n0, add(s, n, sset)))

     weighedTime(Wt(t, n, sset)) = t
```

```
     totalTp(Wt(t, n, sset)) = n

     if(T, wt, wt0) = wt
     if(F, wt, wt0) = wt0
     if(T, sset, sset0) = sset
     if(F, sset, sset0) = sset0

     calculateWeight(p, Wt(t, n, sset)) =
       cwIterate(p, t, n, sset, emptyStreamDataSet)

     cwIterate(p, t, n, emptyStreamDataSet, sset) = Wt(t, n, sset)
     cwIterate(p, t, n, add(s, n0, sset0), sset)      =
       cwIterate(p, plus(t, R(mlty(weightFactor(p, s), n0) ,1))), plus(n, n0),
         sset0, add(s, n0, sset))

     update(p, Wt(t, n, sset), s) = Wt(t, n, update(p, sset, s))

     update(p, emptyStreamDataSet, s) = emptyStreamDataSet   % error
     update(p, add(s, n, sset), s0)   =
       if(eq(s, s0),
         if(cumulative(p, s),
            add(s, plus(n, 1), sset), % car: every detect counts
            add(s, 1, sset)),         % pedestrian: only first detect counts
         add(s, n, update(p, sset, s0)))

sort WaitingSet
func add                 : Pid # Time # Int # WaitingSet    -> WaitingSet
     emptyWaitingSet     :                                  -> WaitingSet
     insertionSort       : WaitingSet                       -> WaitingSet
     insert              : Pid #  Time # Int # WaitingSet   -> WaitingSet
     in                  : Pid # WaitingSet                 -> Bool
     peel                : WaitingSet                       -> PidSet
     if                  : Bool # WaitingSet # WaitingSet   -> WaitingSet
var  wl, ws, wset, wset0: WaitingSet
     t, t0                : Time
     p, p0                : Pid
     n, n0                : Int
rew  if(T, wset, wset0) = wset
     if(F, wset, wset0) = wset0

     % insertion descending sort on dynamic time and absolute priority
     insertionSort(emptyWaitingSet)   =
       emptyWaitingSet
     insertionSort(add(p, t, n, wset)) =
       insert(p, t, n, insertionSort(wset))

     insert(p, t, n, emptyWaitingSet)      =
       add(p, t, n, emptyWaitingSet)
     insert(p, t, n, add(p0, t0, n, wset))  =
       if(or(leq(t, t0), and(eq(t, t0), leq(priority(p), priority(p0)))),
         add(p0, t0, n, insert(p, t, n, wset)),
         add(p, t, n, add(p0, t0, n, wset)))

     peel(emptyWaitingSet) =  emptyPidSet
     peel(add(p, t, n, wset)) = add(p, peel(wset))

sort PendingSet
func add                : Pid # Time # PendingSet             -> PendingSet
```

```
     emptyPendingSet   :                                   -> PendingSet
     newCs             : PidSet # PendingSet               -> PidSet
     newNcs            : PendingSet                        -> PendingSet
     realizations      : PendingSet # WaitingSet           -> PendingSet
     rIterate          : PendingSet # WaitingSet # PidSet  -> PendingSet
     r1 : PendingSet # Pid # WaitingSet # PidSet # Time
                                                            -> PendingSet
     r2 : PendingSet # Pid # WaitingSet # PidSet # Time # Time # Time
                                                            -> PendingSet
     conflict          : Pid # PendingSet                   -> Bool
     spareTime         : Pid # PendingSet # PidSet          -> Time
     maxWaitingTime    : PendingSet # PidSet                -> Time
     maxWaitingTime    : PendingSet # PidSet # Time         -> Time
     maxTime           : Pid # PendingSet # Time            -> Time
     if                : Bool # PendingSet # PendingSet     -> PendingSet
var  ncs, pset, pset0  : PendingSet
     t, t0, spare, clear: Time
     p, p0              : Pid
     cs, notUsed, pidset: PidSet
     n, n0              : Int
     ws, wset           : WaitingSet
     s                  : Sid
rew  if(T, pset, pset0) = pset
     if(F, pset, pset0) = pset0

     newCs(cs, emptyPendingSet) = cs
     newCs(cs, add(p, t, pset))   =
       if( leq(t, controlCycleTime),
         add(p, newCs(cs, pset)),
         newCs(cs, pset))

     newNcs(emptyPendingSet) = emptyPendingSet
     newNcs(add(p, t, pset))   =
       if( leq(t, controlCycleTime),
         newNcs(pset),
         add(p, sbtr(t, controlCycleTime), newNcs(pset)))

     realizations(ncs, ws) = rIterate(ncs, insertionSort(ws), emptyPidSet)

     rIterate(ncs, add(p, t, n, wset), notUsed) =
       if(conflict(p, ncs),
         rIterate(ncs, wset, add(p, notUsed)),
         if(conflict(p, notUsed),
           % second order realization
           r2(ncs, p, wset, notUsed, R(n,1), plus(plus(R(n,1), yellowTime(p)),
               clearanceDuration(p)), spareTime(p, ncs, notUsed)),
           % first order realization
           r1(ncs, p, wset, notUsed, max(R(n,1), minGreenTime(p)))))

     r1(ncs, p, wset, notUsed, t) =
       add(p, t, rIterate(
           add(p, plus(plus(t, yellowTime(p)), clearanceDuration(p)), ncs),
             wset, notUsed))

     r2(ncs, p, wset, notUsed, t, clear, spare) =
       if(lt(spare, minGreenTime(p)),
         % not enough time to be realized
         rIterate(ncs, wset, add(p, notUsed)),
```

```
        % time available
        if(lt(spare, clear),
          % realize as long as allowed
          add(p, sbtr(sbtr(spare, yellowTime(p)), clearanceDuration(p)),
            rIterate(add(p, spare, ncs), wset, notUsed)),
          % realize as long as wanted
          add(p, t,
            rIterate(add(p, clear, ncs), wset, notUsed))))

    conflict(p, emptyPendingSet) = F
    conflict(p, add(p0, t, pset))  =
      or(conflict(p, p0), conflict(p, pset))

    % find time first phase of notUsed frustrated by p is no longer
    % frustrated by ncs
    spareTime(p, ncs, notUsed) =
      maxWaitingTime(ncs, selectConflict(p, notUsed))

    % find time first phase in pidset is no longer frustrated by ncs
    maxWaitingTime(ncs, emptyPidSet)  = R(0, 1)
      % error: regular use by realizations guarantees non-empty pidSet
    maxWaitingTime(ncs, add(p, pidset)) =
      maxWaitingTime(ncs, pidset, maxTime(p, ncs, R(0, 1)))

    maxWaitingTime(ncs, emptyPidSet, t)  = t
    maxWaitingTime(ncs, add(p, pidset), t) =
      maxWaitingTime(ncs, pidset, min(maxTime(p, ncs, R(0, 1)), t))

    % find time p is no longer frustrated by (any phase in) ncs
    maxTime(p, emptyPendingSet, t) = t
    maxTime(p, add(p0, t0, ncs), t)   =
      if(conflict(p, p0),
        maxTime(p, ncs, max(t, t0)),
        maxTime(p, ncs, t))

% End Dynamic Accounting Functions Definitions


% Begin Process Specifications

act  detect                                : Sid
     setGreen, setYellow, setRed           : Pid
     sendWeighedTime, receiveWeighedTime, commWeighedTime
                                           : Pid # Time # Int
     sendRealization, receiveRealization, commRealization
   : Pid # Time

comm sendWeighedTime  | receiveWeighedTime  = commWeighedTime
     sendRealization  | receiveRealization  = commRealization

proc PclearedRed(p: Pid, wt: WeighedTimeTable , w: Time) =
       integral( [and(leq(R(0,1), v), lt(v, w))],
         sum(s: Sid, detect(s)[v]
                     . PclearedRed(p, update(p, wt, s), sbtr(w, v))
       ))
     + integral( [and(leq(R(0,1), v), lt(v, w))],
         sendWeighedTime(p, weighedTime(wt), totalTp(wt))[v]
         . PclearedRed(p, wt, sbtr(w, v))
```

```
        )
      + integral( [and(leq(R(0,1), v), lt(v, w))],
        sum(t: Time, receiveRealization(p, t)[v]
            . setGreen(p) . Pgreen(p, t)
        ))
      + tau[w] . PclearedRed(p, calculateWeight(p, wt), cycleTime)

      PnonClearedRed(p: Pid, wt: WeighedTimeTable, w: Time) =
        integral( [and(leq(R(0,1), v), lt(v, w))],
          sum(s: Sid, detect(s)[v]
                      . PnonClearedRed(p, update(p, wt, s), sbtr(w, v))
        ))
      + tau[w] . PclearedRed(p, calculateWeight(p, wt), cycleTime)

      Pgreen(p: Pid, t: Time) = setYellow(p)[t] . Pyellow(p)
      Pyellow(p: Pid) = setRed(p)[yellowTime(p)]
        . PnonClearedRed(p, initStreams(p), clearanceDuration(p))

      % CP stands for Control Process

      CPreceive(add(p: Pid, cs: PidSet): PidSet,
                ncs: PendingSet, ws: WaitingSet) =
          sum(greenTime: Time, sum(trafficParticipants: Int,
            receiveWeighedTime(p, greenTime, trafficParticipants)[R(0,1)] )
            . CPreceive(cs, ncs, add(p, greenTime, trafficParticipants, ws) ) )

      CPreceive(emptyPidSet: PidSet, ncs: PendingSet, ws: WaitingSet) =
          CPsend(peel(ws), ncs, realizations(ncs, ws))

      CPsend(cs:PidSet, ncs:PendingSet,
              add(p:Pid, t:Time, ps:PendingSet) : PendingSet) =
          sendRealization(p, t)[R(0,1)] . CPsend(extract(p, cs),
           add(p, plus(plus(t, yellowTime(p)), clearanceDuration(p)), ncs), ps)
      CPsend(cs:PidSet, ncs:PendingSet, emptyPendingSet: PendingSet) =
          tau[controlCycleTime]
          . CPreceive(newCs(cs, ncs), newNcs(ncs), emptyWaitingSet)

      % RTRS stands for Realtime Traffic Regulation System

      RTRS =
        hide
          (    {commRealization, commWeighedTime},
          encap
            (  {sendWeighedTime, receiveWeighedTime,
                sendRealization, receiveRealization},

             CPreceive(actualIntersectionPidSet, emptyPendingSet,
                                                   emptyWaitingSet)
             ||
                     merge(p : actualIntersectionPidSet,
                     PclearedRed(p, initStreams(p), cycleTime))
             )
           )
% End Process Specifications
```