A Car Registration Authority, a Concise PSF-Specification

Inge Bethke

Alban Ponse

University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

E-mail: `inge@fwi.uva.nl - alban@fwi.uva.nl`

Abstract

A PSF specification of a Car Registration Authority is given. Typical is the use of encapsulation of actions having certain data values, and data parametric communication and recursion. The PSF specification has been simulated.

Key Words & Phrases: Specification Language, PSF, ADT (Abstract Data Types), Process Algebra.
1985 Mathematics Subject Classification: 68Q40.
1987 CR Categories: D.1.3, D.2.1, F.1.2.

Contents

1 Introduction **1**

2 PSF specifications **2**
- 2.1 Data in the specification . . . 2
- 2.2 Process part . . . 7
- 2.3 Simulation . . . 12

3 General comments **12**
- 3.1 Comments on the PSF specifications . . . 12
- 3.2 Features not used . . . 13
- 3.3 Features not modelled . . . 13

1 Introduction

The purpose of this work is to describe a systematic and easy to understand PSF [MV90] specification of a "Car Registration Authority". The Car Registration Authority (CRA) models the production and sale of cars. Our specification of CRA is reminiscent to the one described by Wieringa [Wie90]. The origins of CRA are described by van Griethuysen in the ISO 1982 report [Gri82]. The specification described here has been simulated using the PSF toolkit [Vel93].

The tasks of CRA as specified in this paper are

- to register who was the owner of a car at any moment from production to destruction of the car, and
- to monitor certain laws, for example regarding production of cars and transfer of ownership.

For the first task we define various databases in PSF. The second task is essentially modelled by data-parametric actions that originate from process definitions mainly using (data-parametric) encapsulation and communication (synchronisation). After the occurrence of such an action, the process identifier is called recursively with an update of its arguments (which represent the current value of the database).

We tried to come up with a systematic kind of modelling, and certainly did *not* attempt to give a smart or compact specification. On the contrary, some careful study may give the reader inspiration for descriptive or algorithmic 'optimisations'. Furthermore, we specified on the one hand little too much, as to provide a uniform setting in which our specification can be easily extended, and on the other hand not the complete CRA example. The latter is a consequence both of shortcomings of the present version of PSF, and of the decision to only specify the most characteristic features of CRA.

The plan of this paper is as follows. In Section 2.1 we introduce the data part of our specification, and in Section 2.2 the processes operating on these data. Section 2.3 contains a simulation trace obtained with tools developed for PSF. In Section 3.1 we comment on the PSF specification of CRA as defined in Section 2, in particular on the way data and processes interact. In Section 3.2 we comment on what is specified too much, and in Section 3.3 on what kind of specifications are left out.

2 PSF specifications

This section is divided in three parts. First we specify the data of our modelling in PSF, meanwhile explaining their functionality. Then we continue with the specifications of the processes involved, ending with a specification of a process '`Globalcontrol`' that defines (our version) of CRA. Finally, we depict a simulation of `Globalcontrol`.

2.1 Data in the specification

We start off describing the data module `Booleans` from the so called PSF library [vWa93], comprising the sort `BOOLEAN` and (as usual)

- the constants `true` and `false`,
- the unary operation `not`, and
- the binary prefix operations `eq`, `and` and `or`.

As we have employ for a conditional, we import the module `Booleans` into a data module `Conditionals` defining a conditional function `if` by two equations:

```
data module Conditionals
begin
  exports
    begin
      functions
        if: BOOLEAN # BOOLEAN # BOOLEAN   -> BOOLEAN
    end
  imports
    Booleans
  variables
    b,b': -> BOOLEAN
  equations
    [C1] if(true,b,b') = b
    [C2] if(false,b,b') = b'
end Conditionals
```

We also incorporate from the standard library for PSF the data module `Naturals` which defines the Natural numbers and most standard operations. Of the latter, we only use the operations `eq` (the characteristic function for equality), `+` and `lt` (less than). The representation of natural numbers in this data module has a special format: for example the number 15 is represented as `nat(^1^5)`, where the (unary and infix binary) functions `^` yield digit-strings, and the function `nat` converts these to natural numbers, thus providing access to most standard operations.

In order to specify the inhabitants of various (finite) sets in a uniform way, we introduce a separate data module `Items`. This module defines an infinite collection of `items` based on naturals. The normal forms of sort `ITEM` are of the form `i(nat(^0)), i(nat(^1)), i(nat(^2)), i(nat(^3)),` .... Note that the functions `eq`, `n + nat(^1)` and `lt` from the `Naturals` are used to define their counterparts in `Items`.

```
data module Items
begin
  exports
    begin
      sorts
        ITEM
      functions
        i:    NATURAL     -> ITEM
        eq:   ITEM # ITEM -> BOOLEAN
        less: ITEM # ITEM -> BOOLEAN
        s:    ITEM        -> ITEM      -- s: successor function on items
    end
  imports
    Naturals
  variables
    n,m: -> NATURAL
  equations
    [I1] eq(i(n),i(m)) = eq(n,m)
    [I2] less(i(n),i(m)) = lt(n,m)     -- lt: less than
    [I3] s(i(n)) = i(n + nat(^1))
end Items
```

We use the modules `Conditionals` and `Items` to define the data module `Sets`, in which the basic structure underlying the databases is specified. A `SET` is a finite set of enumerated items. `empty-set`, `ins`, `del` and `el` denote the empty set, the usual operations that insert an item into and delete an item from a set, respectively, and the characteristic function of elementhood. Observe that `SET`s are strictly ordered by the equations `[S3]` and `[S4]`.

```
data module Sets
begin
  exports
    begin
      sorts
        SET
      functions
        empty-set:                -> SET
        ins:       ITEM # SET -> SET
        del:       ITEM # SET -> SET
        el:        ITEM # SET -> BOOLEAN
    end
  imports
    Conditionals, Items
  functions
    if: BOOLEAN # SET # SET   -> SET
  variables
    i,j:  -> ITEM
    s,s': -> SET
    b,b': -> BOOLEAN
  equations
    [S1] if(true,s,s') = s
    [S2] if(false,s,s') = s'
    [S3] ins(i,ins(i,s)) = ins(i,s)
    [S4] ins(i,ins(j,s)) = ins(j,ins(i,s)) when less(i,j)=true
    [S4] del(i,empty-set) = empty-set
    [S5] del(i,ins(j,s)) = if(eq(i,j),del(i,s),ins(j,del(i,s)))
    [S6] el(i,empty-set) = false
    [S7] el(i,ins(j,s)) = if(eq(i,j),true,el(i,s))
end Sets
```

Before we present the further PSF specifications of the data part of our modelling, we give an intuitive explanation of their functionality, and depict all internal dependencies (specified by the import sections). In the data modules `Cars`, `Manufacturers` and `Owners` the crucial sorts of cars, car manufacturers and car owners are specified. The data module `Cardb` (Car database) remembers the cars that have been produced and registers when a car has been destructed. As yet, we have no employ for this database. The data module `Ownerdb` collects, during the life span of a car, its owners. We define a data module `Manufacturerdb` for modelling a manufacturer database. This database collects the manufacturers that have been given a permission and registers when a permission has been withdrawn. The data modules introduced above depend on each other via the import structure depicted in Figure 1.

Next we elaborate on how the various databases are defined.

In the data modules `Cars`, `Manufacturers` and `Owners` the sorts of cars, car manufacturers and car owners are specified by importing renamed versions of `Items` and `Sets`. These sorts have respective normal forms `c(nat(^0))`, ..., `m(nat(^0))`, ... and `o(nat(^0))`, ...

```
data module Cars              | data module Manufacturers      | data module Owners
begin                         | begin                          | begin
  imports                     |   imports                      |   imports
    Sets                      |     Sets                       |     Sets
      {renamed by             |       {renamed by              |       {renamed by
         [ITEM -> CAR,        |          [ITEM -> MANF,        |          [ITEM -> OWNER,
          SET  -> CARS,       |           SET  -> MANFS,       |           SET  -> OWNERS,
          i    -> c]}         |           i    -> m]}          |           i    -> o]}
end Cars                      | end Manufacturers              | end Owners
```

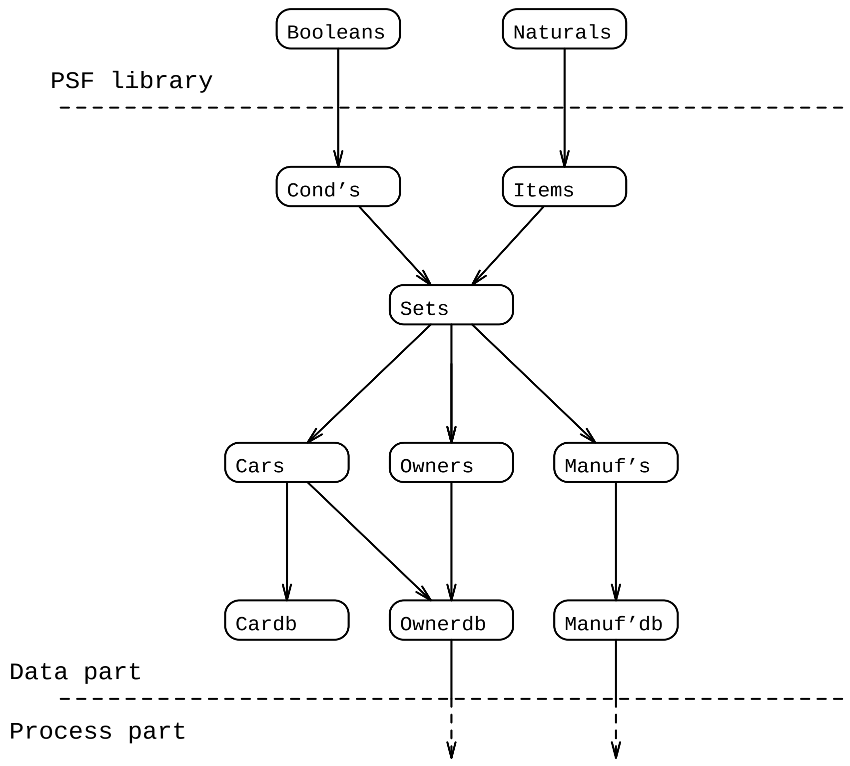

Figure 1: import structure of the data part

In the data module `Cardb` the car database is defined. A car database remembers the cars that have been produced and registers when a car has been destructed using the attribute function `set-exist`. The constant `empty-cdb` denotes the empty database, and the function `set-exist` adds an entry of the form (`c,true`) to the car database in case the car `c` has been produced, and an entry of the form (`c,false`) in case `c` has been destroyed. The function `exist` tests whether or not a certain car is still 'existent'. Finally, the function `cars` collects the set of all cars that are recorded as either existent or destroyed in the database.

```
data module Cardb
begin
  exports
    begin
      sorts
       CDB
      functions
        empty-cdb:                      -> CDB
        set-exist: CDB  # CAR # BOOLEAN -> CDB
        exist:     CDB  # CAR           -> BOOLEAN
        cars:      CDB                  -> CARS
    end
  imports
    Cars
```

```
  variables
    c,c': -> CAR
    cdb:  -> CDB
    b:    -> BOOLEAN
    cs:   -> CARS
  equations
    [CDB1] exist(empty-cdb,c) = false
    [CDB2] exist(set-exist(cdb,c',true),c) = if(eq(c,c'),true,exist(cdb,c))
    [CDB3] exist(set-exist(cdb,c',false),c) = if(eq(c,c'),false,exist(cdb,c))
    [CDB4] cars(empty-cdb) = empty-set
    [CDB5] cars(set-exist(cdb,c,b)) = ins(c,cars(cdb))
end Cardb
```

We define a data module `Manufacturerdb` for modelling a manufacturer database. This database collects (in the same manner as the car database can handle cars) the manufacturers that have been given a permission and registers when a permission has been with-drawn. The attribute function `set-perm` adds an entry of the form `(m,true)` to the manufacturer database in case a manufacturer `m` has been given permission, and adds an entry `(m,false)` in case the permission of `m` has been withdrawn. The function `perm` collects the current 'permission status' of a particular manufacturer `m` as a Boolean value, returning `true` whenever the last entry was of the form `(m,true)` and `false` in all other cases. Note that the function `manfs` collects the set of manufacturers that are in the database, irrespective of their permission status.

```
data module Manufacturerdb
begin
  exports
    begin
      sorts
        MDB
      functions
        empty-mdb:                      -> MDB
        set-perm: MDB  # MANF # BOOLEAN -> MDB
        perm:     MDB  # MANF           -> BOOLEAN
        manfs:    MDB                   -> MANFS
    end
  imports
    Manufacturers
  variables
    m,m': -> MANF
    mdb:  -> MDB
    b:    -> BOOLEAN
    ms:   -> MANFS
  equations
    [MDB1] perm(empty-mdb,m) = false
    [MDB2] perm(set-perm(mdb,m',true),m) = if(eq(m,m'),true,perm(mdb,m))
    [MDB3] perm(set-perm(mdb,m',false),m) = if(eq(m,m'),false,perm(mdb,m))
    [MDB4] manfs(empty-mdb) = empty-set
    [MDB5] manfs(set-perm(mdb,m,b)) = ins(m,manfs(mdb))
end Manufacturerdb
```

The owner database defined in the data module `Ownerdb` collects, during the life span of a car, its owners. The attribute function `set-owner` adds an entry of the form `(o,c)` to the owner database in case an owner `o` has obtained a car `c`. Dumping a car is a final event: a car that has been dumped cannot be owned or manufactured again. The function `close` is used to specify this. The function `owns` tests whether a certain owner possesses a certain car, and `owners` collects the set of owners that are latest registered in the database as owning a car (note that `owners(close(odb,c)) = owners(odb)` by equation [ODB6]). The functions `cars` and `all-cars` collect the set of cars that have been registered, where the former only collects the cars currently owned, and the latter collects all cars that have been owned (and possibly dumped).

```
data module Ownerdb
begin
  exports
    begin
      sorts
        ODB
      functions
        empty-odb:                    -> ODB
        set-owner: ODB # OWNER # CAR -> ODB
        close:     ODB # CAR         -> ODB
        owns:      ODB # OWNER # CAR -> BOOLEAN
        owners:    ODB               -> OWNERS
        cars:      ODB               -> CARS
        all-cars:  ODB               -> CARS
    end
  imports
    Cars, Owners
  variables
    c,c': -> CAR
    o,o': -> OWNER
    odb:  -> ODB
  equations
    [ODB1] owns(empty-odb,o,c) = false
    [ODB2] owns(set-owner(odb,o',c'),o,c) =
                       if(eq(c,c'),if(eq(o,o'),true,false),owns(odb,o,c))
    [ODB3] owns(close(odb,c'),o,c) = if(eq(c,c'),false,owns(odb,o,c))
    [ODB4] owners(empty-odb) = empty-set
    [ODB5] owners(set-owner(odb,o,c)) = ins(o,owners(odb))
    [ODB6] owners(close(odb,c)) = owners(odb)
    [ODB7] cars(empty-odb) = empty-set
    [ODB8] cars(set-owner(odb,o,c)) = ins(c,cars(odb))
    [ODB9] cars(close(odb,c)) = del(c,cars(odb))
    [ODB10] all-cars(empty-odb) = empty-set
    [ODB11] all-cars(set-owner(odb,o,c)) = ins(c,all-cars(odb))
    [ODB12] all-cars(close(odb,c)) = ins(c,all-cars(odb))
end Ownerdb
```

2.2 Process part

In addition, we define five process modules:

- `Carcontrol` - Cars can be transferred or destructed.
- `Permcontrol` - Permissions can be given or with-drawn.
- `Manufacturercontrol` - Manufacturers can (repeatedly) obtain and lose a permission. Provided a permission is obtained, cars can be produced.
- `Ownercontrol` - Owners can buy (new or used) cars they do not own. They can also dump a car.
- `Globalcontrol` - In this process module the concurrent activity resulting from that defined in the four modules above is defined.

The two process modules mentioned first are independent (contain no imports). They both define processes that provide the remaining other processes with suitable communication partners.

In the process module `Carcontrol` a process `Car` is defined that repeatedly chooses between an action `destruct` and an action `transfer`. This gives other control processes the possibility to use either of these to communicate to respectively `last-` and `next-owner`ship actions.

```
process module Carcontrol
begin
  exports
    begin
      atoms
        destruct
        transfer
      processes
        Car
    end
  definitions
    Car = (transfer + destruct).Car
end Carcontrol
```

In the process module `Permcontrol` a process `Perm` is defined that repeatedly chooses between actions `give-perm` and and `withdraw-perm`, giving other control processes the possibility to use either of these to communicate to respectively `grant-` and `suspend-perm` actions. As is also the case for the previous module, this behaviour can easily be redefined to one that is reactive to either external, not yet specified control processes, or to internal constraints. For example, one can use counters to satisfy the constraint that never more than five permissions at once are given out.

```
process module Permcontrol
begin
  exports
    begin
      atoms
        give-perm
        withdraw-perm
      processes
        Perm
    end
  definitions
    Perm = (give-perm + withdraw-perm).Perm
end Permcontrol
```

The process modules defined in this section constitute the import structure depicted in Figure 2. In particular, both `Manufacturercontrol` and `Ownercontrol` employ the databases defined in the previous section.

The activities concerning manufacturers are modelled in a process module `Manufacturercontrol`. A manufacturer can, if he has a permission, either lose that permission or produce cars. If he does not have a permission, he can receive one. The loss of a permission is not a final event: a manufacturer who has lost his permission can receive one again. The two processes `Manf` defined in this module specify what actions can occur. The 'initial' process `Manf` (without parameters) instantiates the encapsulated version of `Manf(empty-mdb,empty-set)`. The first argument represents (the current value of) the manufacturer database, the second keeps control of the already existing manufacturers and what new manufacturers may (systematically by successorship) be added to the database. Initially the process `Manf(empty-mdb,empty-set)` can only perform a `receive-perm(m(nat(^0)),true)` action, after which it continues with the encapsulation of
`Manf(set-perm(empty-mdb,m(nat(^0)),true),ins(m(nat(^0)),manfs( mdb)))`.
At this point the second equation of the parametric process `Manf` specifies further behaviour: either the process `Manf(set-perm(empty-mdb,m(nat(^0)),true),empty-set)` is invoked by recursive descendence, or an action `receive-perm(s(m(nat(^0))),true)` can pass the encapsulation and the

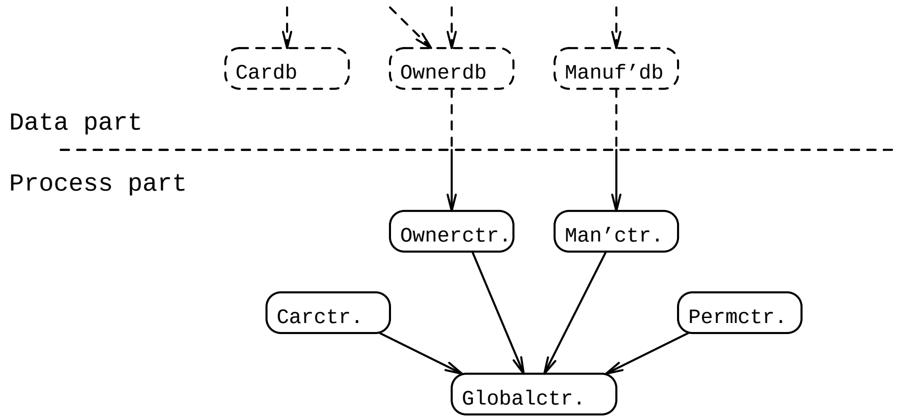

Figure 2: import structure of the process part

recursion updates both parameters of `Manf`. In the first case, either `m(nat(^0))` may lose its permission, or it may start to produce cars. After one of these actions has occurred, the process continues with its arguments appropriately updated. Inspection of the process definitions of `Manf` reveals the further possibilities.

```
process module Manufacturercontrol
begin
  exports
    begin
      atoms
        receive-perm: MANF # BOOLEAN
        lose-perm:    MANF # BOOLEAN
        produce:      MANF # BOOLEAN
      processes
        Manf
    end
  imports
    Manufacturerdb
  processes
    Manf: MDB # MANFS
  sets of
    atoms
      H = {receive-perm(m,false),lose-perm(m,false),
           produce(m,false) | m in MANF}
  variables
    m:   -> MANF
    ms:  -> MANFS
    mdb: -> MDB
  definitions
    Manf(mdb,empty-set) = receive-perm(m(nat(^0)),not(perm(mdb,m(nat(^0))))).
                          Manf(set-perm(mdb,m(nat(^0)),true),ins(m(nat(^0)),manfs(mdb)))
                          + lose-perm(m(nat(^0)),perm(mdb,m(nat(^0)))).
                          Manf(set-perm(mdb,m(nat(^0)),false),manfs(mdb))
                          + produce(m(nat(^0)),perm(mdb,m(nat(^0)))).Manf(mdb,manfs(mdb))

    Manf(mdb,ins(m,ms)) = Manf(mdb,ms)
                          + receive-perm(s(m),not(perm(mdb,s(m)))).
                          Manf(set-perm(mdb,s(m),true),ins(s(m),manfs(mdb)))
```

```
                         + lose-perm(s(m),perm(mdb,s(m))).
                         Manf(set-perm(mdb,s(m),false),manfs(mdb))
                         + produce(s(m),perm(mdb,s(m))).Manf(mdb,manfs(mdb))
    Manf = encaps(H,Manf(empty-mdb,empty-set))
  end Manufacturercontrol
```

Ownership is specified in the module `Ownercontrol`. Owners can buy newly produced cars and used cars they do not own. They can also transfer their own cars to a car dump. Dumping a car is a final event: a car that has been dumped cannot be bought again. The process `Owner`, defining ownership takes three arguments. The owner database, the first argument, can systematically be updated by reference to the current sets of owners and cars (the other two arguments), according to the `buy` and `dump` actions that can be performed.

```
process module Ownercontrol
begin
  exports
    begin
      atoms
        buy-new: OWNER # CAR # BOOLEAN
        buy-old: OWNER # CAR # BOOLEAN
        dump:    OWNER # CAR # BOOLEAN
      processes
        Owner
    end
  imports
    Ownerdb
  processes
    Owner: ODB # OWNERS # CARS
  sets of
    atoms
      H = {buy-new(o,c,false), buy-old(o,c,false), dump(o,c,false) |
           o in OWNER, c in CAR}
  variables
    o:   -> OWNER
    os:  -> OWNERS
    c:   -> CAR
    cs:  -> CARS
    odb: -> ODB
  definitions
    Owner(odb,empty-set,empty-set) =
      buy-new(o(nat(^0)),c(nat(^0)),not(el(c(nat(^0)),all-cars(odb)))).
      Owner(set-owner(odb,o(nat(^0)),c(nat(^0))),
                 ins(o(nat(^0)),owners(odb)),ins(c(nat(^0)),all-cars(odb)))
    + buy-old(o(nat(^0)),c(nat(^0)),and(el(c(nat(^0)),cars(odb)),not(owns(odb,o(nat(^0)),c(nat(^0)))))).
      Owner(set-owner(odb,o(nat(^0)),c(nat(^0))),
                 ins(o(nat(^0)),owners(odb)),all-cars(odb))
    + dump(o(nat(^0)),c(nat(^0)),owns(odb,o(nat(^0)),c(nat(^0)))).
      Owner(close(odb,c(nat(^0))),owners(odb),all-cars(odb))

    Owner(odb,empty-set,ins(c,cs)) =
      Owner(odb,empty-set,cs)
    + buy-new(o(nat(^0)),s(c),not(el(s(c),all-cars(odb)))).
      Owner(set-owner(odb,o(nat(^0)),s(c)),
                  ins(o(nat(^0)),owners(odb)),ins(s(c),all-cars(odb)))
    + buy-old(o(nat(^0)),s(c),and(el(s(c),cars(odb)),not(owns(odb,o(nat(^0)),s(c))))).
      Owner(set-owner(odb,o(nat(^0)),s(c)),
                 ins(o(nat(^0)),owners(odb)),all-cars(odb))
    + dump(o(nat(^0)),s(c),owns(odb,o(nat(^0)),s(c))).
      Owner(close(odb,s(c)),owners(odb),all-cars(odb))

    Owner(odb,ins(o,os),empty-set) =
      Owner(odb,os,empty-set)
```

```
    + buy-new(s(o),c(nat(^0)),not(el(c(nat(^0)),all-cars(odb)))).
      Owner(set-owner(odb,s(o),c(nat(^0))),
                ins(s(o),owners(odb)),ins(c(nat(^0)),all-cars(odb)))
    + buy-old(s(o),c(nat(^0)),and(el(c(nat(^0)),cars(odb)),not(owns(odb,s(o),c(nat(^0)))))).
      Owner(set-owner(odb,s(o),c(nat(^0))),
                ins(s(o),owners(odb)),all-cars(odb))
    + dump(s(o),c(nat(^0)),owns(odb,s(o),c(nat(^0)))).
      Owner(close(odb,c(nat(^0))),owners(odb),all-cars(odb))

    Owner(odb,ins(o,os),ins(c,cs)) =
      Owner(odb,os,ins(c,cs))
    + Owner(odb,ins(o,os),cs)
    + buy-new(s(o),s(c),not(el(s(c),all-cars(odb)))).
      Owner(set-owner(odb,s(o),s(c)),
                ins(s(o),owners(odb)),ins(s(c),all-cars(odb)))
    + buy-old(s(o),s(c),and(el(s(c),cars(odb)),not(owns(odb,s(o),s(c))))).
      Owner(set-owner(odb,s(o),s(c)),
                ins(s(o),owners(odb)),all-cars(odb))
    + dump(s(o),s(c),owns(odb,s(o),s(c))).
      Owner(close(odb,s(c)),owners(odb),all-cars(odb))

    Owner = encaps(H,Owner(empty-odb,empty-set,empty-set))
end Ownercontrol
```

In the process module `Globalcontrol`, the low-level processes `Owner`, `Manf`, `Car` and `Perm` are merged. This is also the place where the communications, `first-owner`, `next-owner`, `last-owner`, `grant-perm` and `suspend-perm`, are defined. For example,

```
produce(m,true) | buy-new(o,c,true) = first-owner(c,m,o)
```

indicates that if manufacturer `m` may produce a new car, and owner `o` wants to buy this new car, which is according to the `ownerdb` car `c` (the third argument `true` expresses that this car is brand new indeed), then a communication `first-owner(c,m,o)` may occur.

```
process module Globalcontrol
begin
  exports
    begin
      atoms
        first-owner:  CAR # MANF # OWNER
        next-owner:   CAR # OWNER
        last-owner:   CAR # OWNER
        grant-perm :  MANF
        suspend-perm: MANF
      processes
        Globalcontrol
    end
  imports
    Ownercontrol, Manufacturercontrol, Carcontrol, Permcontrol
  sets of
    atoms
      H = { buy-new(o,c,b), buy-old(o,c,b), dump(o,c,b), produce(m,b),
            destruct, transfer, receive-perm(m,b), lose-perm(m,b), give-perm,
            withdraw-perm | c in CAR, m in MANF, o in OWNER, b in BOOLEAN }
  communications
    produce(m,true) | buy-new(o,c,true) = first-owner(c,m,o)
    for c in CAR, o in OWNER, m in MANF
    buy-old(o,c,true) | transfer = next-owner(c,o)
    for c in CAR, o in OWNER
    dump(o,c,true) | destruct = last-owner(c,o)
    for c in CAR, o in OWNER
```

```
    receive-perm(m,true) | give-perm = grant-perm(m)
    for m in MANF
    lose-perm(m,true) | withdraw-perm = suspend-perm(m)
    for m in MANF
  definitions
    Globalcontrol = encaps(H, Owner || Manf || Car || Perm)
end Globalcontrol
```

2.3 Simulation

In PSF there are various tools that operate on specifications [Vel93]. As an example, we show a small simulation trace of the process `Globalcontrol`. This trace is obtained with the tools `psf` (compilation to the 'tool interface language') and `sim` (for simulation). Observe that in the simulation trace depicted below, at most two manufactures at a time are permitted to produce cars:

```
com. grant-perm(m(nat((^ 0))))
com. first-owner(c(nat((^ 0))), m(nat((^ 0))), o(nat((^ 0))))
com. first-owner(c(nat((^ 1))), m(nat((^ 0))), o(nat((^ 1))))
com. next-owner(c(nat((^ 1))), o(nat((^ 2))))
com. grant-perm(m(nat((^ 1))))
com. first-owner(c(nat((^ 2))), m(nat((^ 1))), o(nat((^ 3))))
com. next-owner(c(nat((^ 1))), o(nat((^ 0))))
com. suspend-perm(m(nat((^ 0))))
com. next-owner(c(nat((^ 0))), o(nat((^ 1))))
com. last-owner(c(nat((^ 0))), o(nat((^ 1))))
com. grant-perm(m(nat((^ 2))))
com. last-owner(c(nat((^ 1))), o(nat((^ 0))))
com. next-owner(c(nat((^ 2))), o(nat((^ 1))))
com. last-owner(c(nat((^ 2))), o(nat((^ 1))))
com. suspend-perm(m(nat((^ 2))))
com. first-owner(c(nat((^ 3))), m(nat((^ 1))), o(nat((^ 2))))
com. last-owner(c(nat((^ 3))), o(nat((^ 2))))
```

3 General comments

In this section we give some comments on the PSF specifications contained in this paper. Furthermore we explain why the redundant specification of the data module `Cardb` is incorporated, and mention a few natural extensions of our modelling.

3.1 Comments on the PSF specifications

We used the following data/process interaction facilities of PSF:

Data-parametric encapsulation. Actions can only occur if they are instantiated with certain data values. For example, the `Manf` process in module `Manufacturercontrol` can only execute an action `receive-perm(m(nat(^0),true)`, but never an action `receive-perm(m(nat(^0),false)` (the second argument is the result of the test whether `m(nat(^0))` already has a permission according to the database `mdb`). Note that this use of encapsulation can be used to mimick *guarded commands* [Dij76, BB91, GP90b] by admitting actions to have a 'guard value' that can be matched by the encapsulation operator.

Data-parametric communication. Communications can only occur between actions that are instantiated with certain data values.

Data-parametric recursion. After the occurrence of a certain action, the concerning process identifier can be updated recursively, again allowing certain actions to occur, and others not.

For a systematic account of data/process interaction, the reader is referred to [MV90, GP91b].

3.2 Features not used

We specified little too much, according to which modules are used by the process `Globalcontrol`: a database for cars is also specified in the module `Cardb` (this database remembers the cars that have been produced and registers when a car has been destructed). This is done to give a uniform setting in which our specification can easily be extended. Conform to [Wie90], different peculiarities of cars can be taken into account, such as information on serial number, registration number, model, fuel consumption and the like.

3.3 Features not modelled

First observe that no "`sell`" actions occur. The act of selling a car is modelled implicitly via `buy` en `transfer` actions. This is caused by the restriction of PSF to the so called *handshaking* paradigm (two-party communication). To date, it is considered to enrich PSF with the feature 'multi-party communication' [Mau93].

The sort `OWNER` can be replaced by for example the two sorts `GARAGE` and `PERSON` so that restricted behaviour can be specified. For example, a person can only buy a car from a garage or another person, and not anymore from a manufacturer. Also the behaviour of a garage can be restricted compared to what is specified for the more abstract kind of owner. For example, a garage can cease to exist if it does not posses any cars. These kind of refinements have been specified by students in a course on Software Engineering at the University of Amsterdam (1992). In this paper we do not give specifications of such refinements, as they are nothing more than small variations on the specification of ownership in this paper.

References

[BB91] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Methods, Proceedings Summer School Marktoberdorf 1991, NATO ASI Series F88*, pages 273–323. Springer-Verlag, 1991.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall International, Englewood Cliff, 1976.

[GP90b] J.F. Groote and A. Ponse. Process algebra with guards. Report CS-R9069, CWI, Amsterdam, 1990. To appear in *Formal Aspects of Computing.*

[GP91b] J.F. Groote and A. Ponse. Proof theory for μCRL. Report CS-R9138, CWI, 1991.

[Gri82] J.J. van Griethuysen. Concepts and terminology for the conceptual schema and the information base. Technical Report ISO TC97/SC5/WG3, 1982.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fund. Inf.*, XII:85-139, 1990.

[Mau93] S. Mauw. Personal communications, 1993.

[Vel93] G.J. Veltink. The PSF toolkit. *Computer Networks and ISDN Systems*, 25:875-898, 1993.

[vWa93] J.J. van Wamel. A library for PSF. Technical Report P9301, Programming Research Group, University of Amsterdam, 1993.

[Wie90] R.J. Wieringa. Algebraic Foundations for dynamic conceptual models. Ph.D Thesis, Free University, Amsterdam, 1990.