Specification of the CFS coherency protocol in LOTOS version 4

Charles Pecheur
INRIA Rhône-Alpes
655 avenue de l'Europe
38330 Montbonnot Saint Martin
FRANCE
Tél: +33-4-76.61.52.98
Fax: +33-4-76.61.52.52
E-mail: Charles.Pecheur@inria.fr

cfs.nw 4.4 - 98/02/19

February 19, 1998

1 Introduction

This document contains the LOTOS specification of the CFS coherency protocol, presented in a litterate programming style. The specification covers access to a single page of a CFS file, and describes both the CFS coherency protocol (see process `Site`) and the real transfer and access to file data (see process `Memory`).

Notational Convention The full LOTOS code is provided, in the form of labelled chunks like the following sample[1]:

1a ⟨*sample* 1a⟩≡

```
(* ... some LOTOS text here ... *)
```

A chunk may contain references to other chunks, to be interpreted as textual inclusion:

1b ⟨*other sample* 1b⟩≡

```
(* ... *)
⟨sample 1a⟩
(* ... *)
```

The LOTOS language is officially defined by the ISO standard 8807 [ISO88]. Tutorials can be found in [BB88, Tur93].

[1] This is produced automatically using N. Ramsey's *Noweb* literate programming system.

Model Generation This specification is intended for model-checking using the CADP validation tools [Gar96]. This has some consequences in the way it is written:

- To limit state space explosion, data types are kept as small as possible. In particular, small sets of constants are often used to model potentially large data domains.
- The behaviour part has a bounded synchronization structure (no recursion over parallelism), and the number of concurrent processes is kept to a minimum.
- Equations are written assuming sequential evaluation (i.e. the first applicable equation is applied). This often allows a drastic reduction of the number of equations, but relies on the particular evaluation strategy used by CADP. It is *not* to be interpreted according to the standard algebraic semantics of LOTOS.

Data Type Syntax Extensions The APERO syntax extensions [Pec96] are used to shorten and clarify the definitions of data types. These notations are not standard LOTOS; a translator is used to expand them into plain LOTOS data type definitions (taking into account the requirements of CADP).

2 Version History

Version 1 First version, based on the automaton found in [Fas96], p. 52, plus [Jac]. Describes the synchronization part of different sites for one page (actual memory transfer is not covered). Different control states are modelled as different LOTOS processes.

Version 2 To tackle state space explosion, the different processes are merged in a single one, with the control state represented explicitly as a data variable.

Version 3 Add modelling of page contents. Since the latest revisions of version 2, model generation is handled compositionally, so we take less care into reducing the number of variables in processes. CAESAR's inefficiency in state representation is eliminated in subsequent minimizations.

Version 4 Drastic housecleaning: all unused processes and definitions removed. Intended for final distribution.

3 Data Types

3.1 Base Domains

Booleans and natural numbers are used throughout.

2 ⟨*data types* 2⟩≡

```
library Boolean, NaturalNumber endlib
```

Defines:
`Bool`, used in chunk 4.
`Nat`, never used.

Each site is identified by an identifier of sort `Site`. This sort is defined as an enumerated type and is iterated upon in model generations; it should be kept as small as possible. This specification is bounded to three different sites.

3a ⟨*data types* 2⟩+≡

```
enumtype        SiteType is
enum            site1,site2,site3 : Site
endtype
```

Defines:
`Site`, used in chunks 5, 7–18, and 23b.

`Val` is the sort of page content. This sort is iterated upon and thus is kept as small as possible, i.e. two different values.

3b ⟨*data types* 2⟩+≡

```
enumtype        ValType is
enum            val1, val2 : Val
endtype
```

Defines:
`Val`, used in chunks 5, 13, and 18.

3.2 Interaction Primitives

`CfsCall` describes the Cfs primitives offered to applications.

3c ⟨*data types* 2⟩+≡

```
enumtype        CfsCallType is
enum            read, beginwrite, endwrite : CfsCall
endtype
```

Defines:
`CfsCall`, never used.

`Message` defines the message exchanged between Cfs entities.

3d ⟨*data types* 2⟩+≡

```
enumtype        MessageType is
enum            readrq,readok,writerq,writeok,invalidate,firstmaster : Message
endtype
```

Defines:
`Message`, used in chunks 5b, 12b, and 13.

`State` is used in monitoring interactions, to observe the internal state of the different sites. The last four are transient states where internal information is processed; no message or request can be received in those states.

- `master` The site is master, no one is writing.
- `writing` The site is master and in a writing session.
- `invalid` The site has no valid copy.
- `valid` The site owns a valid copy.
- `waitread` The site is waiting for a valid copy.
- `waitwrite` The site is waiting for mastership.
- `flushrqs` The site is master and is flushing pending requests (transient).
- `forwardrqs` The site has no valid copy and is forwarding pending requests to the current master (transient).
- `invalwriting` The site is invalidating remote copies before writing (transient).
- `invalinvalid` The site is invalidating remote copies while giving up mastership (transient).

4 ⟨*data types* 2⟩+≡

```
enumtype        StateType is
enum            master,writing,invalid,valid,waitread,waitwrite,
                flushrqs,forwardrqs,invalwriting,invalinvalid : State
endtype

type            StateOpns is stateType
opns            istransient : State -> Bool
                ismaster : State -> Bool
eqns forall s : State
ofsort Bool
  istransient(flushrqs) = true ;
  istransient(forwardrqs) = true ;
  istransient(invalinvalid) = true ;
  istransient(invalwriting) = true ;
  istransient(s) = false ;

  ismaster(master) = true ;
  ismaster(writing) = true ;
  ismaster(s) = false ;
endtype
```

Defines:
`State`, used in chunk 6.
Uses `Bool` 2.

3.3 State Variables

SiteSet defines sets of site identifiers, used by a page master to remember all remote copy requesters and holders.

5a ⟨*data types* 2⟩+≡

```
  csettype        SiteSetType is SiteType
  cset            SiteSet
  elements        site1,site2,site3 : Site
  endtype
```

Defines:
  SiteSet, used in chunks 5d and 6.
Uses Site 3a 6.

PktList defines a list of (Site, Message) pairs. It is used by the underlying communication channel to store transitting messages. the Site is the remote (i.e. non-master) site; it can be either the source or the destination of the message, depending on the message type.

5b ⟨*data types* 2⟩+≡

```
  recordtype      PktType is SiteType, Messagetype
  record          pkt : Pkt
  fields          site : Site
                  msg : Message
  endtype

  listtype        PktListType is PktType
  list            PktList
  elements        Pkt
  endtype
```

Defines:
  Pkt, never used.
  PktList, used in chunks 5d and 6.
Uses Message 3d and Site 3a 6.

ValArray is an array of Val indexed on Site, used in process Memory to store the different copies of a page for each site.

5c ⟨*data types* 2⟩+≡

```
  arraytype       ValArrayType is ValType, SiteType
  array           ValArray
  elements        Val
  indices         site1,site2,site3 : Site
  endtype
```

Defines:
  ValArray, used in chunks 5d and 13.
Uses Site 3a 6 and Val 3b.

Some complementary constants for convenience.

5d ⟨*data types* 2⟩+≡

```
  type            ConstantsType is SiteSetType, PktListType, ValArrayType
  opns            nocopies : -> SiteSet
                  norqs : -> PktList
                  init : -> Val
                  init : -> ValArray
  eqns
  ofsort SiteSet
    nocopies = {} ;
  ofsort PktList
    norqs = <> ;
  ofsort Val
    init = val1 ;
  ofsort ValArray
    init = fill(init of Val) ;
  endtype
```

Uses `PktList` 5b, `SiteSet` 5a, `Val` 3b, and `ValArray` 5c.

4 System Processes

4.1 Cfs entity

The process `Site` describes the management of a single page by a Cfs site. This is a state-oriented specification, originally based on the state machine presented in [Fas96]. All state is specified as data parameters. The parameter `state:StateType` encodes the control part of the state.

As a special case, the first site to request (read or write) access to the page receives initial mastership. This is modelled as a `firstmaster` message received *before* the `readrq` or `writerq` has been sent.

Note: for simplification, initial mastership assignment is not covered in the generated models. Instead, mastership is given arbitrarily to `site1`..

6 ⟨*processes* 6⟩≡

```
  process Site [cfsreq,cfsans,send,rcv]
    ( s : Site,
      state : State,
      copies : SiteSet,
      rqs : PktList )
    : noexit :=

    ( ⟨local read 8b⟩ )

    []

    ( ⟨local beginwrite 8c⟩ )

    []

    ( ⟨local endwrite 9a⟩ )
  []

  ( ⟨remote readrq 9b⟩ )

  []

  ( ⟨remote writerq 9c⟩ )

  []

  ( ⟨remote readok 10a⟩ )

  []

  ( ⟨remote writeok 10b⟩ )

  []

  ( ⟨remote invalidate 10c⟩ )

  []

  ( ⟨transient flushrqs 11c⟩ )

  []

  ( ⟨transient forwardrqs 12a⟩ )

  []

  ( ⟨transient invalwriting 11a⟩ )

  []

  ( ⟨transient invalinvalid 11b⟩ )

endproc
```

Defines:
`Site`, used in chunks 5, 7–18, and 23b.
Uses `PktList` 5b, `SiteSet` 5a, and `State` 4.

`InitSite` defines a site in initial state, i.e. no valid copy and both lists are empty. Maybe this should not be used since there is a risk that CAESAR keeps its variables in the state vector.

7 ⟨*processes* 6⟩+≡

```
process InitSite [cfsreq,cfsans,send,rcv] ( s : Site ) : noexit :=

  Site [cfsreq,cfsans,send,rcv] (s,invalid,nocopies,norqs)

endproc
```

Defines:

`InitSite`, used in chunk 19.
Uses `Site` 3a 6.

`InitMaster` is similar to `InitSite`, except that the site is given mastership.

8a ⟨*processes* 6⟩+≡

```
process InitMaster [cfsreq,cfsans,send,rcv] ( s : Site ) : noexit :=

  Site [cfsreq,cfsans,send,rcv] (s,master,nocopies,norqs)

endproc
```

Defines:
`InitMaster`, used in chunk 19.
Uses `Site` 3a 6.

4.1.1 Local Requests

The following paragraphs detail the handling of CFS requests from local applications.

local read

8b ⟨*local read* 8b⟩≡

```
[(state eq master) or (state eq valid) or (state eq invalid)] ->
cfsreq !s !read;
( [state eq master] ->
  cfsans !s !read;
  Site [cfsreq,cfsans,send,rcv] (s,master,copies,rqs)

  []

  [state eq valid] ->
  cfsans !s !read;
  Site [cfsreq,cfsans,send,rcv] (s,valid,copies,rqs)

  []

  [state eq invalid] ->
  ( send !s !readrq !s;
    Site [cfsreq,cfsans,send,rcv] (s,waitread,copies,rqs)
    []
    rcv !s !firstmaster !s;
    cfsans !s !read;
    Site [cfsreq,cfsans,send,rcv] (s,master,copies,rqs) ) )
```

Uses `Site` 3a 6.

local beginwrite

8c ⟨*local beginwrite* 8c⟩≡

```
[(state eq master) or (state eq valid) or (state eq invalid)] ->
cfsreq !s !beginwrite;
( [state eq master] ->
  cfsans !s !beginwrite;
```

```
    Site [cfsreq,cfsans,send,rcv] (s,invalwriting,copies,rqs)

    []

    [state eq valid] ->
    send !s !writerq !s;
    Site [cfsreq,cfsans,send,rcv] (s,waitwrite,copies,rqs)

    []

    [state eq invalid] ->
    ( send !s !writerq !s;
      Site [cfsreq,cfsans,send,rcv] (s,waitwrite,copies,rqs)
      []
      rcv !s !firstmaster !s;
      cfsans !s !beginwrite;
      Site [cfsreq,cfsans,send,rcv] (s,writing,copies,rqs) ) )
```

Uses `Site` 3a 6.

local endwrite

9a ⟨*local endwrite* 9a⟩≡

```
  [state eq writing] ->
  cfsreq !s !endwrite;
  cfsans !s !endwrite;
  Site [cfsreq,cfsans,send,rcv] (s,flushrqs,copies,rqs)
```

Uses `Site` 3a 6.

4.1.2 Remote Messages

The following paragraphs detail the handling of CFS protocol messages from remote CFS sites.

remote readrq

9b ⟨*remote readrq* 9b⟩≡

```
  [(state eq master) or (state eq writing)] ->
  rcv !s !readrq ?s1:Site;
  ( [state eq master] ->
    send !s !readok !s1;
    Site [cfsreq,cfsans,send,rcv] (s,master,insert(s1,copies),rqs)

    []

    [state eq writing] ->
    Site [cfsreq,cfsans,send,rcv]
      (s,writing, copies, rqs+pkt(s1,readrq)) )
```

Uses `Site` 3a 6.

remote writerq

9c ⟨*remote writerq* 9c⟩≡

```
  [(state eq master) or (state eq writing)] ->
  rcv !s !writerq ?s1:Site;
  ( [state eq master] ->
    send !s !writeok !s1;
    Site [cfsreq,cfsans,send,rcv] (s,invalinvalid,copies,rqs)

    []

    [state eq writing] ->
    Site [cfsreq,cfsans,send,rcv]
      (s,writing, copies, rqs+pkt(s1,writerq)) )
```

Uses `Site` 3a 6.

remote readok

10a ⟨*remote readok* 10a⟩≡

```
  [state eq waitread] ->
  rcv !s !readok !s;
  cfsans !s !read;
  Site [cfsreq,cfsans,send,rcv] (s,valid,copies,rqs)
```

Uses `Site` 3a 6.

remote writeok

10b ⟨*remote writeok* 10b⟩≡

```
  [state eq waitwrite] ->
  rcv !s !writeok !s;
  cfsans !s !beginwrite;
  Site [cfsreq,cfsans,send,rcv] (s,writing,copies,rqs)
```

Uses `Site` 3a 6.

remote invalidate Note: unexpected reception of `invalidate` is possible in any state other than `valid`. This has been observed as a cause of deadlock of this specification. These cases have been added in the specification; the message is ignored in these cases.

10c ⟨*remote invalidate* 10c⟩≡

```
  [ (state eq valid) or
    (state eq master) or
    (state eq writing) or
    (state eq waitwrite) or
    (state eq waitread) or
    (state eq invalid)] ->
  rcv !s !invalidate !s;
  ( [state eq valid] ->
    Site [cfsreq,cfsans,send,rcv] (s,invalid,copies,rqs)

    []

    [state ne valid] ->
    Site [cfsreq,cfsans,send,rcv] (s,state,copies,rqs) )
```

Uses `Site` 3a 6.

4.1.3 Transient States

The following paragraphs detail the processing done in transient states. Typically this involves flushing some internal list and sending corresponding messages.

transient invalwriting Invalidate remote copies in `copies` before going to `writing`.

11a ⟨*transient invalwriting* 11a⟩≡

```
  [state eq invalwriting] ->
  ( [copies ne nocopies] ->
    send !s !invalidate !min(copies);
    Site [cfsreq,cfsans,send,rcv] (s,invalwriting,butmin(copies),rqs)

    []

    [copies eq nocopies] ->
    Site [cfsreq,cfsans,send,rcv] (s,writing,copies,rqs) )
```

Uses `Site` 3a 6.

transient invalinvalid Invalidate remote copies in `copies` before going to `invalid`.

11b ⟨*transient invalinvalid* 11b⟩≡

```
  [state eq invalinvalid] ->
  ( [copies ne nocopies] ->
    send !s !invalidate !min(copies);
    Site [cfsreq,cfsans,send,rcv] (s,invalinvalid,butmin(copies),rqs)

    []

    [copies eq nocopies] ->
    Site [cfsreq,cfsans,send,rcv] (s,invalid,copies,rqs) )
```

Uses `Site` 3a 6.

transient flushrqs Answer the pending requests in `rqs`.

11c ⟨*transient flushrqs* 11c⟩≡

```
  [state eq flushrqs] ->
  ( [rqs ne norqs] ->
    ( [msg(first(rqs)) eq readrq] ->
      send !s !readok !site(first(rqs));
      Site [cfsreq,cfsans,send,rcv]
        (s, flushrqs, insert(site(first(rqs)),copies), butfirst(rqs))

      []

      [msg(first(rqs)) eq writerq] ->
      send !s !writeok !site(first(rqs));
      Site [cfsreq,cfsans,send,rcv] (s,forwardrqs,copies,butfirst(rqs)) )

    []
    [rqs eq norqs] ->
    Site [cfsreq,cfsans,send,rcv] (s,master,copies,rqs) )
```

Uses `Site` 3a 6.

transient forwardrqs Invalidate remote copies in `copies`, then forward pending requests in `rqs` to the current master.

12a ⟨*transient forwardrqs* 12a⟩≡

```
  [state eq forwardrqs] ->
  ( [copies ne nocopies] ->
    send !s !invalidate !min(copies);
    Site [cfsreq,cfsans,send,rcv] (s,forwardrqs,butmin(copies),rqs)

    []

    [copies eq nocopies] ->
    ( [rqs ne norqs] ->
      send !s !msg(first(rqs)) !site(first(rqs));
      Site [cfsreq,cfsans,send,rcv] (s,forwardrqs,copies,butfirst(rqs))

      []

      [rqs eq norqs] ->
      Site [cfsreq,cfsans,send,rcv] (s,invalid,copies,rqs) ) )
```

Uses `Site` 3a 6.

4.2 Communication Channel

The following processes define the medium through which CFS sites communicate. All events on `send` and `rcv` have the following attributes:

```
send ?s1 : Site ?m : Msg ?s2 : Site
rcv ?s1 : Site ?m : Msg ?s2 : Site
```

`s1` is the site that sends/receives the message; `s2` is the site concerned by the message. The channel ignores `s1` and keeps `s2`. Note that no destination address is given; each site is responsible for accepting only the messages it is supposed to receive. This works because each kind of message has a well-defined destination: requests go to the master, responses go to the concerned site.

`OutputCell` is a one-slot bounded buffer whose input is restricted to a single site. The restriction to a single message avoids state space explosion. Using a different channel for each site allows messages from different sites to be received in any order (and blows up the state space). This is necessary for a correct working of the protocol; deadlocks have been observed in models with a single common channel.

12b ⟨*processes* 6⟩+≡

```

  process OutputCell [send,rcv] (s : Site) : noexit :=

    send !s ?m:Message ?s1:Site;
    rcv ?dest:Site !m !s1;
    OutputCell [send,rcv] (s)
  endproc
```

Defines:
`OutputCell`, used in chunks 21 and 23b.
Uses `Message` 3d and `Site` 3a 6.

4.3 Memory

`Memory` holds the data (of sort `Val`) of the page controlled through the Cfs protocol. Different copies are kept for each site. The Cfs messages are seen through gate `ctrl` and cause data to be transfered on `readok` and `writeok` messages. Gates `read` and `write` model the access to memory by the application, with the following profiles:

```
   read ?s : Site ?v : Val
   write ?s : Site ?v : Val
```

13 ⟨*processes* 6⟩+≡

```
  process Memory [read,write,ctrl] (mems: ValArray) : noexit :=

    ( choice s:Site []
      read !s !get(s, mems) ;
      Memory [read,write,ctrl] (mems) )

    []

    write ?s:Site ?v:Val;
    Memory [read,write,ctrl] (set(s, v, mems))

    []

    ctrl ?s1:Site ?m:Message ?s2:Site;
    ( [(m eq readok) or (m eq writeok)] ->
      Memory [read,write,ctrl] (set(s2, get(s1, mems), mems))
      []
      [(m ne readok) and (m ne writeok)] ->
      Memory [read,write,ctrl] (mems) )

  endproc

  process InitMemory [read,write,send] : noexit :=
      Memory [read,write,send] (init of ValArray)
  endproc
```

Defines:
`InitMemory`, used in chunk 23b.
`Memory`, never used.
Uses `Message` 3d, `Site` 3a 6, `Val` 3b, and `ValArray` 5c.

5 Environment processes

This section defines processes which describe the expected behaviour of the environment of components of a Cfs system. These processes are used to filter out impossible execution paths when generating those components separately, in a compositional approach.

5.1 Environment for Sites

`MasterSiteProxy`, `SlaveSiteProxy` abstract the behaviour of another site, as seen from a given site through gates `send` and `rcv`. `MasterSiteProxy` covers messages to and form a master site, independently of its number; `SlaveSiteProxy` covers messages to and from a given slave site.

14 ⟨*processes* 6⟩+≡

```
process MasterSiteProxy [send,rcv] (s:Site) : noexit :=

  send !s !readrq !s;
  MasterSiteProxy [send,rcv] (s)

  []

  send !s !writerq !s;
  MasterSiteProxy [send,rcv] (s)

  []

  rcv !s !readok !s;
  MasterSiteProxy [send,rcv] (s)

  []

  rcv !s !writeok !s;
  MasterSiteProxy [send,rcv] (s)

  []

  rcv !s !invalidate !s;
  MasterSiteProxy [send,rcv] (s)

endproc

process SlaveSiteProxy [send,rcv] (s:Site, other:Site) : noexit :=

  rcv !s !readrq !other;
  ( send !s !readok !other;
    SlaveSiteProxy [send,rcv] (s,other)
    []
    send !s !readrq !other;
    SlaveSiteProxy [send,rcv] (s,other) )

  []

  rcv !s !writerq !other;
  ( send !s !writeok !other;
    SlaveSiteProxy [send,rcv] (s,other)
    []
```

```
      send !s !writerq !other;
      SlaveSiteProxy [send,rcv] (s,other) )

    []

    send !s !invalidate !other;
    SlaveSiteProxy [send,rcv] (s,other)

  endproc
```

Defines:
`MasterSiteProxy`, used in chunk 15a.
`SlaveSiteProxy`, used in chunk 15a.
Uses `Site` 3a 6.

To constitute an environment for a given site, we need a single `MasterSiteProxy` plus one `SlaveSiteProxy` for each site. It is not necessary to include a `SlaveSiteProxy` for the constrained site, because in no case can a site become its own master: it cannot receive a `readrq` or `writerq` from itself, nor need to send an `invalidate` to itself.

15a ⟨*processes* 6⟩+≡

```
  process Site2Proxy [send,rcv] (s:Site, other:Site) : noexit :=
    MasterSiteProxy [send,rcv] (s)
    |||
    SlaveSiteProxy [send,rcv] (s,other)
  endproc

  process Site3Proxy [send,rcv]
      (s:Site, other1:Site, other2:Site) : noexit :=
    MasterSiteProxy [send,rcv] (s)
    |||
    SlaveSiteProxy [send,rcv] (s,other1)
    |||
    SlaveSiteProxy [send,rcv] (s,other2)
  endproc
```

Defines:
`Site2Proxy`, used in chunk 20a.
`Site3Proxy`, used in chunk 20a.
Uses `MasterSiteProxy` 14, `Site` 3a 6, and `SlaveSiteProxy` 14.

5.2 Environment for Channels

Note: since `Site` and `Message` are small enumerated types, it is possible to generate the graph for a finite channel without any constraint.

`SlaveSendProxy`, `MasterSendProxy` fix the messages sent by a site on its output channel, resp. in slave and master state. Note that the former depends only on the sender while the latter also depends on the receiver. They are used for restricting the environment of channel processes.

15b ⟨*processes* 6⟩+≡

```
  process SlaveSendProxy [send] (s:Site) : noexit :=

    send !s !readrq !s;
    SlaveSendProxy [send] (s)

    []

    send !s !writerq !s;
    SlaveSendProxy [send] (s)

  endproc

  process MasterSendProxy [send] (s:Site, other:Site) : noexit :=

    send !s !readok !other;
    MasterSendProxy [send] (s,other)

    []

    send !s !writeok !other;
    MasterSendProxy [send] (s,other)

    []

    send !s !readrq !other;
    MasterSendProxy [send] (s,other)

    []

    send !s !writerq !other;
    MasterSendProxy [send] (s,other)

    []

    send !s !invalidate !other;
    MasterSendProxy [send] (s,other)

  endproc
```

Defines:
  `MasterSendProxy`, used in chunk 17.
  `SlaveSendProxy`, used in chunk 17.
Uses `Site` 3a 6.

`RcvProxy` fixes message received from some channel by another site. It is used for restricting the environment of channel processes.

16 ⟨*processes* 6⟩+≡

```
  process RcvProxy [rcv] (s:Site, other:Site) : noexit :=

    rcv !other !readrq ?z:site;
    RcvProxy [rcv] (s,other)

    []

    rcv !other !writerq ?z:site;
    RcvProxy [rcv] (s,other)

    []

    rcv !other !readok !other;
    RcvProxy [rcv] (s,other)

    []

    rcv !other !writeok !other;
    RcvProxy [rcv] (s,other)

    []

    rcv !other !readrq !other;
    RcvProxy [rcv] (s,other)

    []

    rcv !other !writerq !other;
    RcvProxy [rcv] (s,other)

    []

    rcv !other !invalidate !other;
    RcvProxy [rcv] (s,other)

  endproc
```

Defines:
`RcvProxy`, used in chunk 17.
Uses `Site` 3a 6.

Channel proxies are grouped to constrain a given channel, according to the expected number of sites. With the same reasoning as for site proxies, we can safely omit communications from a site to itself.

17 ⟨*processes* 6⟩+≡

```
  process Channel2Proxy [send,rcv]
    (s:Site, other:Site) : noexit :=
    SlaveSendProxy [send] (s)
    |||
    MasterSendProxy [send] (s,other)
    |||
    RcvProxy [rcv] (s,other)
  endproc

  process Channel3Proxy [send,rcv]
    (s:Site, other1:Site, other2:Site) : noexit :=
    SlaveSendProxy [send] (s)
    |||
    MasterSendProxy [send] (s,other1)
    |||
    MasterSendProxy [send] (s,other2)
    |||
    RcvProxy [rcv] (s,other1)
    |||
    RcvProxy [rcv] (s,other2)
  endproc
```

Uses `MasterSendProxy` 15b, `RcvProxy` 16, `Site` 3a 6, and `SlaveSendProxy` 15b.

5.3 User behaviour

Process `GeneralUser` links calls to CFS and accesses to memory. It encodes the expected use of CFS by the application:

- call (request/answer) `read` then read the page any number of times;
- call `beginwrite` and `endwrite` before and after writing and/or reading the page any number of times.

18 ⟨*processes* 6⟩+≡

```
  process GeneralUser [read,write,cfsreq,cfsans] (s:Site) : noexit :=

    cfsreq !s !read;
    cfsans !s !read;
    ReadingUser [read,write,cfsreq,cfsans] (s)

    []

    cfsreq !s !beginwrite;
    cfsans !s !beginwrite;
    WritingUser [read,write,cfsreq,cfsans] (s)

  endproc

  process ReadingUser [read,write,cfsreq,cfsans] (s:Site) : noexit :=

    read !s ?v:Val;
    ReadingUser [read,write,cfsreq,cfsans] (s)

    []

    GeneralUser [read,write,cfsreq,cfsans] (s)

  endproc

  process WritingUser [read,write,cfsreq,cfsans] (s:Site) : noexit :=

    read !s ?v:Val;
    WritingUser [read,write,cfsreq,cfsans] (s)
    []
    write !s ?v:Val;
    WritingUser [read,write,cfsreq,cfsans] (s)

    []

    cfsreq !s !endwrite;
    cfsans !s !endwrite;
    GeneralUser [read,write,cfsreq,cfsans] (s)

  endproc
```

Defines:
  `GeneralUser`, used in chunk 23.
  `ReadingUser`, never used.
  `WritingUser`, never used.
Uses `Site` 3a 6 and `Val` 3b.

6 Instanciated Processes

This section defines instances of previously defined processes as parameter-less processes. They are used with CAESAR's `-root` option to generate models of system components in a compositional approach.

Site instances

19 ⟨*processes* 6⟩+≡

```
  process Site1 [cfsreq,cfsans,send,rcv] : noexit :=
    InitSite [cfsreq,cfsans,send,rcv] (site1)
  endproc

  process Site2 [cfsreq,cfsans,send,rcv] : noexit :=
    InitSite [cfsreq,cfsans,send,rcv] (site2)
  endproc

  process Site3 [cfsreq,cfsans,send,rcv] : noexit :=
    InitSite [cfsreq,cfsans,send,rcv] (site3)
  endproc

  process Master1 [cfsreq,cfsans,send,rcv] : noexit :=
    InitMaster [cfsreq,cfsans,send,rcv] (site1)
  endproc

  process Site12 [cfsreq,cfsans,send,rcv] : noexit :=
    Master1 [cfsreq,cfsans,send,rcv]
    |||
    Site2 [cfsreq,cfsans,send,rcv]
  endproc

  process Site123 [cfsreq,cfsans,send,rcv] : noexit :=
    Master1 [cfsreq,cfsans,send,rcv]
    |||
    Site2 [cfsreq,cfsans,send,rcv]
    |||
    Site3 [cfsreq,cfsans,send,rcv]
  endproc
```

Uses `InitMaster` 8a and `InitSite` 7.

Proxy instances

20a ⟨*processes* 6⟩+≡

```
  process Proxy12 [send,rcv]  : noexit :=
    Site2Proxy [send,rcv] (site1,site2)
  endproc

  process Proxy21 [send,rcv]  : noexit :=
    Site2Proxy [send,rcv] (site2,site1)
  endproc

  process Proxy123 [send,rcv] : noexit :=
    Site3Proxy [send,rcv] (site1,site2,site3)
  endproc

  process Proxy213 [send,rcv] : noexit :=
    Site3Proxy [send,rcv] (site2,site1,site3)
  endproc

  process Proxy312 [send,rcv] : noexit :=
    Site3Proxy [send,rcv] (site3,site1,site2)
  endproc
```

Uses `Site2Proxy` 15a and `Site3Proxy` 15a.

Site instances with proxies

20b ⟨*processes* 6⟩+≡

```
  process Site1With2 [cfsreq,cfsans,send,rcv] : noexit :=
    Site1 [cfsreq,cfsans,send,rcv]
    |[send,rcv]|
    Proxy12 [send,rcv]
  endproc

  process Site2With1 [cfsreq,cfsans,send,rcv] : noexit :=
    Site2 [cfsreq,cfsans,send,rcv]
    |[send,rcv]|
    Proxy21 [send,rcv]
  endproc

  process Site1With23 [cfsreq,cfsans,send,rcv] : noexit :=
    Site1 [cfsreq,cfsans,send,rcv]
    |[send,rcv]|
    Proxy123 [send,rcv]
  endproc

  process Site2With13 [cfsreq,cfsans,send,rcv] : noexit :=
    Site2 [cfsreq,cfsans,send,rcv]
    |[send,rcv]|
    Proxy213 [send,rcv]
  endproc

  process Site3With12 [cfsreq,cfsans,send,rcv] : noexit :=
    Site3 [cfsreq,cfsans,send,rcv]
    |[send,rcv]|
    Proxy312 [send,rcv]
  endproc

  process Master1With2 [cfsreq,cfsans,send,rcv] : noexit :=
    Master1 [cfsreq,cfsans,send,rcv]
    |[send,rcv]|
    Proxy12 [send,rcv]
  endproc

  process Master1With23 [cfsreq,cfsans,send,rcv] : noexit :=
    Master1 [cfsreq,cfsans,send,rcv]
    |[send,rcv]|
    Proxy123 [send,rcv]
  endproc
```

Cell instances

21 ⟨*processes* 6⟩+≡

```
  process OutputCell1 [send,rcv] : noexit :=
    OutputCell [send,rcv] (site1)
  endproc

  process OutputCell2 [send,rcv] : noexit :=
    OutputCell [send,rcv] (site2)
  endproc

  process OutputCell3 [send,rcv] : noexit :=
    OutputCell [send,rcv] (site3)
  endproc

  process OutputCell12 [send,rcv] : noexit :=
    OutputCell1 [send,rcv]
    |||
    OutputCell2 [send,rcv]
  endproc

  process OutputCell123 [send,rcv] : noexit :=
    OutputCell1 [send,rcv]
    |||
    OutputCell2 [send,rcv]
    |||
    OutputCell3 [send,rcv]
  endproc
```

Uses `OutputCell` 12b.

Channel proxy instances

22a ⟨*processes* 6⟩+≡

```
  process ChannelProxy12 [send,rcv]  : noexit :=
    Channel2Proxy [send,rcv] (site1,site2)
  endproc

  process ChannelProxy21 [send,rcv]  : noexit :=
    Channel2Proxy [send,rcv] (site2,site1)
  endproc

  process ChannelProxy123 [send,rcv] : noexit :=
    Channel3Proxy [send,rcv] (site1,site2,site3)
  endproc

  process ChannelProxy213 [send,rcv] : noexit :=
    Channel3Proxy [send,rcv] (site2,site1,site3)
  endproc

  process ChannelProxy312 [send,rcv] : noexit :=
    Channel3Proxy [send,rcv] (site3,site1,site2)
  endproc
```

Cell instances with proxies

22b ⟨*processes* 6⟩+≡

```
  process OutputCell1with2 [send,rcv] : noexit :=
    OutputCell1 [send,rcv]
    |[send,rcv]|
    ChannelProxy12 [send,rcv]
  endproc

  process OutputCell2with1 [send,rcv] : noexit :=
    OutputCell2 [send,rcv]
    |[send,rcv]|
    ChannelProxy21 [send,rcv]
  endproc

  process OutputCell1with23 [send,rcv] : noexit :=
    OutputCell1 [send,rcv]
    |[send,rcv]|
    ChannelProxy123 [send,rcv]
  endproc

  process OutputCell2with13 [send,rcv] : noexit :=
    OutputCell2 [send,rcv]
    |[send,rcv]|
    ChannelProxy213 [send,rcv]
  endproc

  process OutputCell3with12 [send,rcv] : noexit :=
    OutputCell3 [send,rcv]
    |[send,rcv]|
    ChannelProxy312 [send,rcv]
  endproc
```

General User instances

23a ⟨*processes* 6⟩+≡

```
  process GeneralUser1 [read,write,cfsreq,cfsans] : noexit :=
    GeneralUser [read,write,cfsreq,cfsans] (site1)
  endproc

  process GeneralUser2 [read,write,cfsreq,cfsans] : noexit :=
    GeneralUser [read,write,cfsreq,cfsans] (site2)
  endproc

  process GeneralUser3 [read,write,cfsreq,cfsans] : noexit :=
    GeneralUser [read,write,cfsreq,cfsans] (site3)
  endproc
```

Uses `GeneralUser` 18.

7 Top Level specification

Note: the models used for the validation of CFS have been generated compositionally, using the instanciated processes above to produce separate components. The following top-level behaviour is given for illustration only; currently it cannot be compiled monolithically within available memory.

The specification covers the management of and access to a single page by three concurrent sites. An initial `firstmaster` message is generated spontaneously before the channel starts its normal operation.

23b ⟨*behaviour* 23b⟩≡

```
  (
    GeneralUser [read,write,cfsreq,cfsans] (site1)
    |||
    GeneralUser [read,write,cfsreq,cfsans] (site2)
    |||
    GeneralUser [read,write,cfsreq,cfsans] (site3)
  )
  |[read,write,cfsreq,cfsans]|
  (
    (
      Initsite [cfsreq,cfsans,send,rcv] (site1)
      |||
    Initsite [cfsreq,cfsans,send,rcv] (site2)
    |||
    Initsite [cfsreq,cfsans,send,rcv] (site3)
  )
  |[send,rcv]|
  (
    ( rcv ?s1:Site !firstmaster ?s2:Site;
      (
        OutputCell [send,rcv] (site1)
        |||
        OutputCell [send,rcv] (site2)
        |||
        OutputCell [send,rcv] (site3)
      )
    )
    |[send]|
    InitMemory [read,write,send]
  )
)
```

Uses `GeneralUser` 18, `InitMemory` 13, `OutputCell` 12b, and `Site` 3a 6.

Finally, here is the specification itself.

24 ⟨*cfs.LOTOS* 24⟩≡

```
(*****************************************************************
  Compiled from @(#)cfs.nw      4.4 - 98/02/19
  Charles Pecheur, INRIA Rhone-Alpes
*****************************************************************)

specification CfsSystem [cfsreq,cfsans,send,rcv,read,write] : noexit

  ⟨data types 2⟩

behaviour

  ⟨behaviour 23b⟩

where

  ⟨processes 6⟩

endspec
```

This code is written to file `cfs.LOTOS`.

References

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.

[Fas96] Jean-Philippe Fassino. *Utilisation d'une mémoire virtuelle répartie pour le support d'un système de fichiers réparti.* DEA, Université Joseph Fourier, Grenoble, June 1996.

[Gar96] Hubert Garavel. An Overview of the Eucalyptus Toolbox. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, pages 76–88. University of Maribor, Slovenia, June 1996.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[Jac] Thierry Jacquin. Le protocole de cohérence mémoire de CFS. Personal notes.

[Pec96] Charles Pecheur. *Improving the Specification of Data Types in* LOTOS. Doctorate thesis, University of Liège, November 1996. Collection of Publications of the Faculty of Applied Sciences, Nr 171.

[Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques – An Introduction to ESTELLE, LOTOS, and SDL.* John Wiley, 1993.

Index of LOTOS Identifiers