INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Advanced Modelling and Verification Techniques Applied to a Cluster File System

Charles Pecheur

N° 3416

May 1998

THÈME 1

Rapport de recherche

ISSN 0249-6399

Advanced Modelling and Verification Techniques Applied to a Cluster File System

Charles Pecheur*

Thème 1 — Réseaux et systèmes
Projet VASY

Rapport de recherche n3416 — May 1998 — 55 pages

Abstract:
This report describes the application of elaborated formal modelling techniques and tools from the CADP toolset for LOTOS to the validation of CFS, a distributed file system. After a short overview of the LOTOS specification of CFS, we describe the techniques used for model generation and validation, and their application to CFS. Two original aspects are put forth: firstly, the model is generated in a compositional way, by putting together separately generated sub-components; secondly, the extensible, data-aware temporal logic checker XTL is used to express and validate properties of the system. In particular, an XTL extension providing richer diagnostics is presented. The full commented LOTOS specification is provided in appendix.

Key-words: Formal Method, Specification, Model Checking, Distributed File System, Compositional Generation, Temporal Logic, LOTOS, XTL.

(Résumé : tsvp)

Short version of this report in Charles Pecheur, "Advanced Modelling and Verification Techniques Applied to a Cluster File System", submitted for publication.

* Charles.Pecheur@inria.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Techniques avancées de modélisation et de vérification appliquées à un système de gestion de fichiers en grappe

Résumé :

Ce rapport relate l'application de techniques et d'outils de modélisation formelle élaborés, appartenant à la boîte à outils CADP pour LOTOS, pour la validation du système de fichiers réparti CFS. Après un court aperçu de la spécification de CFS en LOTOS, nous décrivons les techniques utilisées pour générer et valider les modèles, et leur application à CFS. Deux aspects originaux sont mis en évidence : premièrement, le modèle est généré de manière compositionnelle, en assemblant des composants générés séparément; deuxièmement, le vérificateur de logique temporelle XTL, extensible et supportant les données, est utilisé pour exprimer et valider les propriétés du système. En particulier, on présente une extension de XTL fournissant des diagnostics enrichis. La spécification LOTOS complète et commentée est fournie en appendice.

Mots-clé : Méthode Formelle, Spécification, Vérification de Modèles, Système de Fichiers Réparti, Génération Compositionnelle, Logique Temporelle, LOTOS, XTL.

1 Introduction

The benefits of formal methods for the design of complex distributed systems are now widely acknowledged. Many formalisms, algorithms and tools have been proposed for formally describing concurrent applications, expressing their properties and automating their verification. Two main approaches have been extensively studied: *theorem proving* and *model checking*. The latter, while applicable only to systems with a finite state space, offers the advantage of requiring much less participation from the user.

One should not conclude that model checking reduces to writing a specification and calling the checker, though. The well-known state space explosion is always lurking, and significant results only come out from the combination of large computing resources, sophisticated tools and skilled formal method experts.

This report illustrates the use of advanced techniques for the modelling and verification of CFS (Cluster File System) [Fas96], a distributed file system built on top of the ARIAS shared memory architecture [DHMdP96]. Two original aspects are put forth:

- the use of *compositional model generation* [FKM93], in order to produce a model that would have been impossible to generate in a single step, and
- the use of the *extensible temporal logic checker* XTL [Mat98] and the development of an XTL extension providing richer diagnostics.

The rest of this first section gives a survey of the LOTOS [ISO88] specification language and the CADP/EUCALYPTUS toolset [Gar96], which have been used in this project. Section 2 presents the CFS system along with its specification, Section 3 describes the compositional technique used to generate a model from this specification, and Section 4 discusses the verification task, including the development of an extension for the XTL checker.

1.1 Overview of LOTOS

LOTOS is a standardized Formal Description Technique intended for the specification of communication protocols and distributed systems. Its design was motivated by the need for a language with a high abstraction level and a strong mathematical basis, which could be used for the description and analysis of complex systems. As a design choice, LOTOS consists of two "orthogonal" sub-languages:

The data part is based on the well-known theory of algebraic abstract data types [Gut77], more specifically on the ACT ONE specification language [dMRV92]. Data types are defined using an equational formalism, which we will not present here. Indeed, most data types of our specification are defined in a higher level language, which is automatically translated into plain ACT ONE [Pec96].

The control part is based on a process algebra, combining the best features of CCS [Mil89] and CSP [Hoa85]. A concurrent system is described as a collection of processes interacting by rendez-vous. The behaviour of each process is built compositionally using an algebra of operators (choice, parallel composition, ...) Behaviours can manipulate data values and exchange them through their interactions.

This report does not assume familiarity with LOTOS from the reader; the few forthcoming commented LOTOS excerpts should be self-explanatory. Table 1 describes the main LOTOS operators. Tutorials for LOTOS are available, e.g. [BB88, Tur93].

`stop`	An inactive behaviour (like 0 in arithmetics).
`G !V ?X:S; B`	Interact on gate `G`, sending `V` and receiving a value of sort `S` in `X`, then behave as `B` (other input/output combinations are possible).
`B1 [] B2`	Behave as either `B1` or `B2`, whichever does something first.
`[E] -> B`	If `E` is true then behave as `B`.
`B1 |[G1,..,Gn]| B2`	`B1` in parallel with `B2`, synchronized on gates `G1`, ..., `Gn` (`|||` means no synchronization, `||` means full synchronization).
`hide G1,..,Gn in B`	make actions of `B` on gates `G1`, ..., `Gn` invisible from the outside.
`exit`	Successful termination.
`B1 >> B2`	`B1` followed by `B2`, when `B1` terminates successfully.
`B1 [> B2`	Behave as `B1` until either `B1` terminates or `B2` performs its first action; in the latter case `B1` is discarded.
`P [G1,..,Gn] (V1,..,Vm)`	Call process `P`, with gate and value parameters `G1`, ..., `Gn` and `V1`, ..., `Vm`.

Table 1: Main LOTOS operators

The model (i.e. the meaning) of a LOTOS specification is defined as the graph of all its possible actions (technically, this kind of graph is called a *Labelled Transition System*, or LTS, but we will keep the simpler words "model" and "graph" in this article). Models can be compared according to different equivalence and refinement criteria. In this study, we use observational equivalence [Mil89] for minimization, that is, we reduce models into minimal observationally equivalent ones.

LOTOS has been applied to many complex systems such as network services and protocols [ISO89, L94] but also cryptographic protocols [LBK+96] or hardware architectures [CGM+96]. A number of tools have been developed for LOTOS, covering user needs in such various areas as edition, simulation, compilation, test generation and formal verification.

1.2 The EUCALYPTUS/CADP Toolset

All the work reported in this report has been done within the framework of the EUCALYPTUS LOTOS Toolset [Gar96], an X-Windows based, user-friendly interface federating several complementary LOTOS tools from different sources. An important part of the EUCALYPTUS toolset is CADP (CÆSAR/ALDÉBARAN Development Package) [FGK+96, GJM+97], a leading edge toolbox dedicated to the formal validation of distributed systems.

CADP offers an integrated set of functionalities ranging from interactive simulation to exhaustive, model-based verification methods, and includes sophisticated approaches to deal with large case-studies. In addition to LOTOS, it also supports lower-level formalisms such as finite state machines and networks of communicating automata. In this case study, the following CADP tools were used:

- CÆSAR [GS90] compiles the control part of a LOTOS program into its transition graph. The data part is translated by the CÆSAR.ADT compiler [Gar89] into executable C code, which is used to compute the graph.
- ALDÉBARAN [Fer89] is a verification tool for comparing or minimizing graphs with respect to any of several simulation and bisimulation relations.
- OPEN/CÆSAR [Gar98] is an open programming interface allowing to explore a graph in a controlled way. Several CADP tools, such as XSIMULATOR and GENERATOR described below, are based on this technology. The CÆSAR compiler can produce OPEN/CÆSAR code allowing on-the-fly execution of a LOTOS specification. EXP.OPEN is another OPEN/CÆSAR code producer, giving access to a network of models of communicating processes.
- XSIMULATOR is an interactive program for exploring the behaviour of a LOTOS specification. It allows to walk through the alternative branches of the graph, using back and forth step-by-step execution.
- GENERATOR performs an exhaustive exploration and generates the complete graph of a model. It thus plays a similar role as CÆSAR but can be applied to other OPEN/CÆSAR sources, such as networks of processes in combination with EXP.OPEN.
- XTL [Mat98] is a programmable temporal logic checker, based on a specialized functional programming language equipped with primitives for graph exploration. Definitions of several well-known temporal logics such as ACTL and the modal μ-calculus are provided, and new ones can easily be added. Further details about XTL are given in Section 4.

The EUCALYPTUS toolbox also contains the APERO data type pre-processor [Pec96]. This compiler provides convenient concise syntax extensions for declaring many common families of data structures such as records, enumerations, sets, lists, as well as general ML-style constructor declarations. APERO translates these declarations into standard LOTOS type definitions, equipped with all the usual associated operations (constructors, selectors, equality, etc.).

2 Specification of CFS

2.1 Presentation of ARIAS and CFS

ARIAS [DHMdP96] is a shared memory support system implemented as an extension of the AIX operating system. It provides a virtual memory among a set of machines, in such a way that applications share a unique address space. Rather than using a single coherence protocol that would be expensive and overly restrictive for most applications, ARIAS allows such protocols to be plugged into the system as *specialization modules* according to the needs of specific applications, resulting in better performance. The ARIAS memory space is composed of fixed size *blocks* (called *zones* in ARIAS) that are the smallest units of shared access.

The *Cluster File System* (CFS) [Fas96] is a distributed file system built on top of ARIAS, with the double purpose of validating the ARIAS system itself and experimenting with distributed applications that use shared files as a programming paradigm. The resulting structure is illustrated in Figure 1, where the shaded areas are not covered by the LOTOS specification.

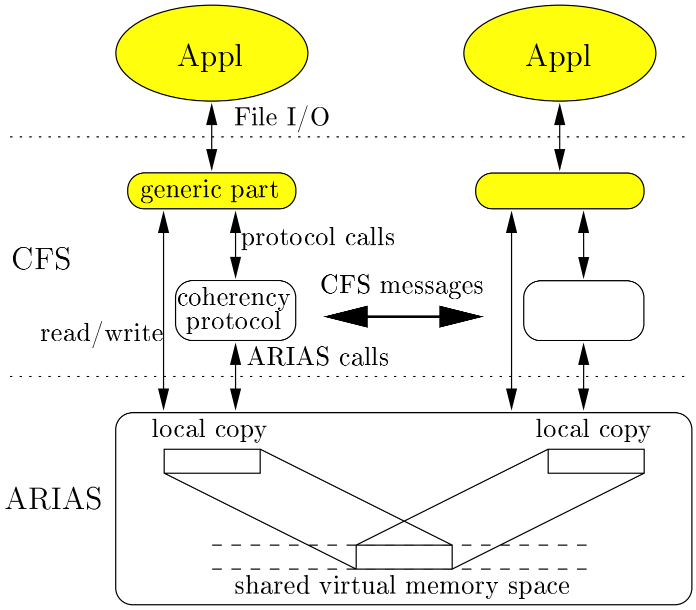

Figure 1: ARIAS and CFS

Several file coherency protocols can co-exist, using different specialization modules. In practice, four coherency protocols have been implemented in Cfs. Among them, the *migratory protocol* is designed to take full advantage of the Arias system and stands out after the multiple benchmarks described in [Fas96]. The specification and verification work presented here focuses on that protocol, which is referred to as the "Cfs protocol" in the sequel. The Cfs protocol relies on the notion of *mastership*, inherited from Arias: at any time, a *master* site owns the reference copy of the block data. Mastership can move between sites during the lifetime of the block; this accounts for much of the flexibility offered by Arias. Every Cfs protocol message goes either from some slave to the master or vice-versa.

2.2 Structure of the Specification

To perform model-based verification, we need to generate a model of finite (and tractable) size. This puts constraints on the Lotos specification: for example, the number of parallel processes must be statically bounded, and choices over infinite ranges are forbidden. Furthermore, various parameters (data ranges, buffer sizes, etc.) are set to minimal values to keep the size of the model within reachable bounds.

The Cfs protocol manages each block of memory independently, so our Lotos specification focuses on the management of a single block (because of this, the block address field in all interactions never changes and is therefore omitted in the specification). Both the Cfs protocol and the Arias service that is used by this protocol are specified. The size of the system is fixed to three sites: this is both an imposed maximum w.r.t. state space explosion and a requested minimum w.r.t. the coverage of possible scenarios in the system (some interesting situations do not occur with only two sites).

Most data types have been defined using the Apero syntax extensions. Besides reducing the size of data type declarations (82 vs. 433 lines of data type definitions), these notations are also much more readable, avoid the burden of equational definitions and hide the technical complications needed to allow the compilation of algebraic data type definitions. The complete specification (with Apero notations) is about 1000 lines long, and is provided in appendix at the end of this report.

The specification models the Cfs protocol at two different levels of abstraction:

- the process `CFS` covers the *control level*, where we consider only the calls to Cfs primitives for acquiring and releasing access to the Cfs block;
- the process `Complete` adds complementary processes to `CFS` to compose the *data level*, where we also take into account access to and modification of the data in the block of memory itself.

The top-level structure of the resulting specification is shown on Figure 2.

The Cfs protocol itself sits at the control level, and is specified for a given site as a process `Site` with four gates: `cfsreq` and `cfsans` support calls from the applications, as request/answer pairs of events, and `send` and `rcv` support emission and reception of Cfs protocol messages through the underlying Arias system. At this level, the behaviour of

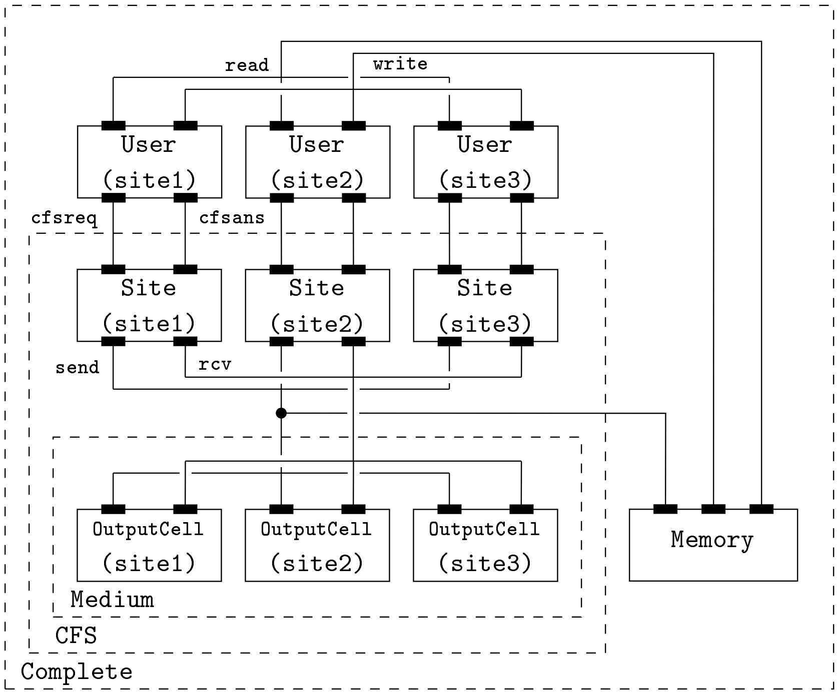

Figure 2: Structure of the LOTOS specification of CFS

ARIAS is reduced to a communication facility, described in the process `Medium`. For modelling purposes, the buffering capacity is limited to one message per sending site: `Medium` is made of single-slot queues `OutputCell` put in parallel, one per site.

The data level part in process `Memory` holds the local copies of the block at each site and applies `read` and `write` events on them. It also observes the messages passing through the `send` gate and propagates master copies accordingly, i.e. following any `readok` or `writeok` message. The size of the model will depend on the cube of the range of possible block values (since there are three sites and thus three local copies): this range has therefore been reduced to only two values.

To complete the picture, a process `User` for each site enforces the correct use of CFS synchronization primitives: for example, `write` events are only allowed between `beginwrite` and `endwrite` CFS calls (this is an abstraction of the shaded upper layer of CFS on Figure 1).

Figure 3 gives an illustration of how these different parts work together.

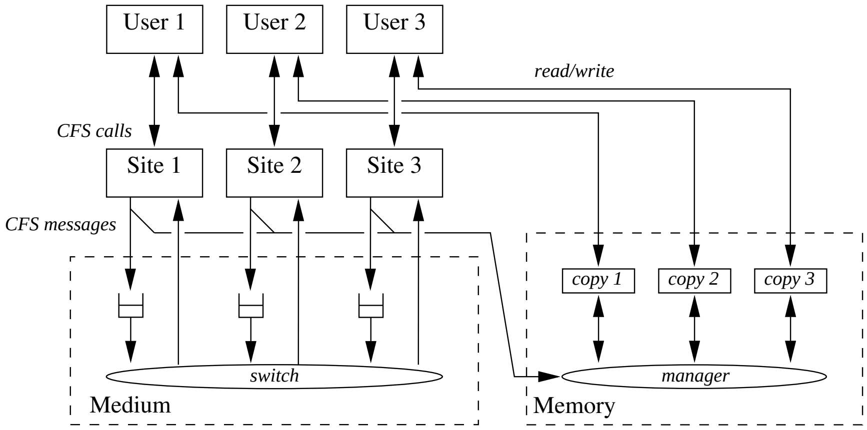

Figure 3: ARIAS service for CFS

2.3 The CFS Protocol

The source CFS definition [Fas96] describes the protocol as an input-output state automaton. The definition of process `Site` essentially captures that automaton, although several rounds of discussions with the designers were needed to reach the level of accuracy needed for formal specification. For illustration, Figure 4 shows the automaton corresponding to the specification: $!m$ (resp. $?m$) stands for *emission* (resp. *reception*) of m, CFS calls are in **bold** (vs. CFS messages), and $\{m\}$ denotes a repetition of m. Deadlocks detected in early models revealed several missing cases in the original description, drawn as dotted edges.

In the LOTOS specification of CFS, the state variables of the protocol become parameters of the LOTOS process. The state itself is also encoded as a parameter (this produces smaller models than defining different mutually recursive processes for each state). The body of `Site` is a non-deterministic choice between reception of CFS requests and messages. For example, the reception of a `readok` message is specified as:

```
process Site [cfsreq,cfsans,send,rcv]
            (s : Site, state : State, ...) : noexit :=
  ... [] (                         (* ... or                    *)
    [state eq waitread] ->         (* if in state waitread then   *)
    rcv !s !readok !s;             (*   receive readok(s)        *)
    cfsans !s !read;               (*   answer pending read       *)
    Site [cfsreq,cfsans,send,rcv]
      (s, valid, ...)              (*   goto state valid          *)
  ) [] ...                         (* or ...                    *)
endproc
```

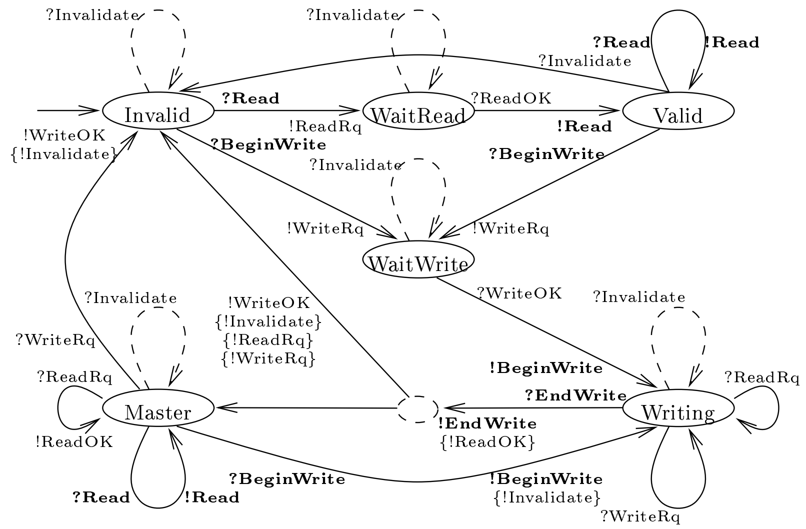

Figure 4: State of a block in the CFS protocol

3 Model Generation

The resulting LOTOS specification of CFS has a finite model, but is too complex to be compiled in a monolithic way using available tools and computers. Indeed, early attempts on the control part alone produced a (essentially unusable) model with 2.7 million states and 9.2 million transitions. Instead of this, we used a divide-and-conquer approach, compiling sub-components of the system separately before combining them together, while minimizing each intermediate model before using it. This section explains this compositional technique and discusses its qualities and limits.

3.1 Tools for Compositional Model Generation

Compositional generation can be handled in CADP through the EXP.OPEN tool. EXP.OPEN takes as input a LOTOS-like behaviour expression using only parallel and hiding operators, describing a network of communicating automata. It allows the exploration of the resulting model by any OPEN/CÆSAR tool. In particular, GENERATOR can be used to produce the explicit representation of that model. Concretely, this appears as a single program invocation, in which object code produced by EXP.OPEN is linked with library code for GENERATOR and executed. Using this technique, compositional generation is obtained through the following steps:

1. generate the model for each component of the system using CÆSAR and minimize it (modulo observational equivalence) using ALDÉBARAN,

2. combine some components using EXP.OPEN, produce the combined graph with GENERATOR and minimize it using ALDÉBARAN,

3. repeat the previous step until obtaining the model of the whole system.

ALDÉBARAN is also able to handle networks of communicating automata, merging the combination and minimization phases. However, usage has shown that using EXP.OPEN and GENERATOR and then minimizing the resulting graph with ALDÉBARAN is more efficient, in both terms of memory and time.

The delicate part of compositional generation, however, is to decide where to cut the whole system into separate components and in which order to combine them. Indeed, two interacting parts $P||Q$ generally strongly constrain each other's behaviour, and generating P or Q separately can produce a much larger graph than $P||Q$ itself, or even an infinite one, thus compromizing the approach.

To overcome this, a solution is to synchronize P with an *environment* E_P, so that $P||E_P$ produces a smaller model that can be substituted for P. This is sound provided that the substitution does not modify the global model. In turn, this is guaranteed if E_P is a *conservative* approximation of the rest of the system as seen from P, i.e. if E_P allows all executions that P can go through as part of the whole system.

Let us mention that CADP provides a tool called PROJECTOR [KM97] that implements a closely related principle and is even able to control the validity of the environment E_P. However, PROJECTOR loops infinitely on some components of the CFS model and thus could not be used here.

3.2 Generation of a Model of CFS

From the specification depicted on Figure 2, a complete model of the management of a single CFS block has been generated compositionally, using environments to reduce the initial sizes of processes `User` and `OutputCell`. Besides models for `Complete` and its components, including `CFS`, The following models have also been generated:

`Abstract` = `Complete` where only gates `read` and `write` remain visible. This gives a very abstract view in terms of values read and written in memory, while assuming (because of `User` processes) that CFS primitives are called appropriately.

`Abstract2` = two concurrent instances of `Abstract`, modelling read/write access to two different blocks. An "address" attribute is added to the events of each instance to distinguish them.

Table 2 gives the sizes of the different components and the generation and minimization times (in seconds, on a Sun Ultra-1 workstation).

Table 2: Model generation statistics

process	gen.	min.	#states	#trans
`Site ||` `EnvSite` (×3)	5.0	0.1	75	130
`OutputCell ||` `EnvOutputCell` (×3)	4.5	<0.1	13	30
`Medium`	3.8	3.1	2,197	15,210
`CFS`	8.0	7.9	11,031	34,728
`User` (×3)	2.0	0.1	6	14
`Memory`	19.9	57.5	8	504
`UserMemory`	9.5	13.7	1,728	103,680
`Complete`	1:57.7	2:39.7	66,324	350,532
`Abstract`	–	5:07:53.2	14	90
`Abstract2`	2.9	2.7	196	2,520

It is worth noting that our simplifications (three sites, single-slot communication channels, two block values) affect dimensions in the system itself only, not the behaviour of its environment (the correlation imposed by `User` processes is part of the definition of CFS). In this sense, we obtain a general model of the system, in contrast with some of our other previous experiments [Pec97] in which only restricted and finite scenarios have been modelled.

The model obtained for process `Abstract` is particularly interesting: it is a highly symmetrical graph with only 14 states, shown in Figure 5 (labels are abbreviated for clarity: R (resp. W) !s !v stands for *read (resp. write) value v from site s*). In states 0 and 11, all local copies have the same value. The three outgoing edges correspond to any of the sites changing that value, and the following internal transitions show the propagation of the new value to other sites.

From a technical point of view, the two copies of `Abstract` combined to produce `Abstract2` were obtained by relabelling the `Abstract` model using the UNIX `sed` utility. This solution is much cheaper than generating each copy separately from the source LOTOS process.

A fully non-synchronized composition as in `Abstract2` is hit headlong by exponential inflation, however: for n independent components, the number of states is $S_n = {S_1}^n$ and the number of transitions is $T_n = n.T_1.{S_1}^{n-1}$. According to this, combining two concurrent instances of `Complete` would go far beyond available computing resources.

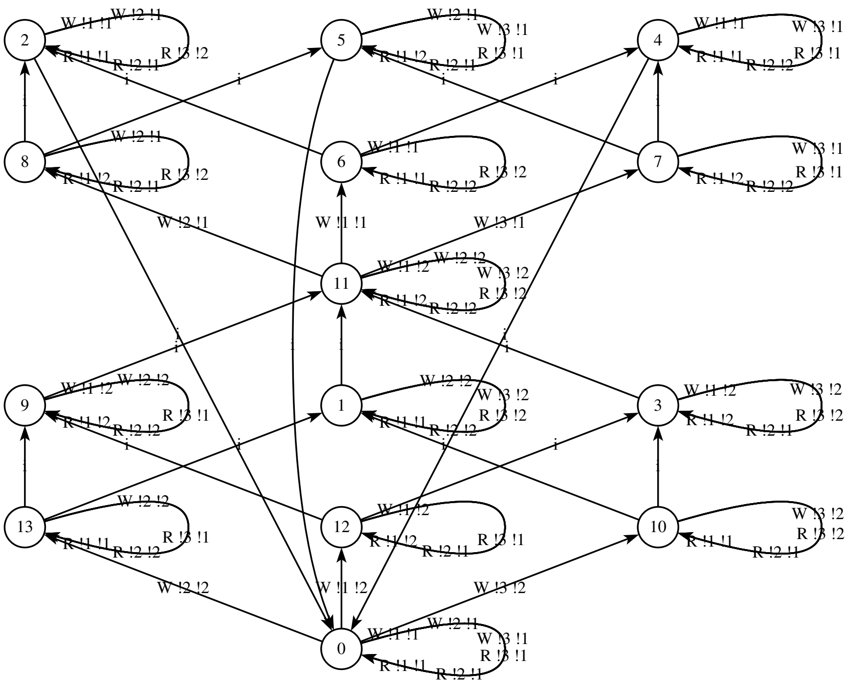

Figure 5: Model of the `Abstract` process

4 Verification

The properties of the CFS protocol have been expressed and evaluated as temporal logic formulas, using the XTL tool. This section gives an overview of XTL, describes how temporal logic operators have been re-defined to obtain more detailed evaluation results, and discusses the verification of the CFS specification using XTL.

4.1 Overview of XTL

Though primarily intended for evaluation of temporal logic formulas, XTL is in full generality a compiler for a functional language applied to a labelled transition system. The XTL language is equipped with data types for states, transitions and labels, and sets thereof, and functions for manipulating them (e.g. initial state, incoming and outgoing transitions of a state, source and target states of a transition). For example, the following XTL expression computes the set of all non-deterministic states:

```
{ S : state where
    exists T1 : edge among out(S), T2 : edge among out(S) where
```

```
        (T1 <> T2) and (label(T1) = label(T2))
    end_exists
}
```

It can also do pattern matching on the labels of transitions and thus access the individual attributes of structured LOTOS events. The following example searches for some transition on `G` with integer attribute larger than 10:

```
exists T : edge where
    T -> [ G ?X : integer where X>10 ]
end_exists
```

Results are reported using a side-effect `print` function. Temporal operators are defined as functions and/or macros using these primitives; definitions for standard logics (e.g. HML, ACTL, modal *mu*-calculus) are provided as XTL libraries. We have used (a fragment of) the ACTL logic [NV90], which has four primitive temporal operators (besides usual boolean connectors):

$$F ::= \mathsf{EX}_A F \mid \mathsf{AX}_A F \mid \mathsf{E}[F\ {}_A\mathsf{U}\ F] \mid \mathsf{A}[F\ {}_A\mathsf{U}\ F]$$

The following table gives their meaning, their XTL syntax, and introduces a few other derived operators used in this report:

XTL syntax	math syntax	definition	meaning
`EX_A(`A`,`F`)`	$\mathsf{EX}_A F$	(primitive)	Some path does an A step that reaches F.
`AX_A(`A`,`F`)`	$\mathsf{AX}_A F$	(primitive)	All paths do an A step that reaches F.
`EU_A(`F`,`A`,`G`)`	$\mathsf{E}[F\ {}_A\mathsf{U}\ G]$	(primitive)	Some path stays in F through A steps until it reaches G.
`AU_A(`F`,`A`,`G`)`	$\mathsf{A}[F\ {}_A\mathsf{U}\ G]$	(primitive)	All paths stay in F through A steps until they reach G.
`Dia(`A`,`F`)`	$\langle A\rangle F$	$\mathsf{EX}_A F$	Some A step reaches F.
`Box(`A`,`F`)`	$[A]F$	$\neg\mathsf{EX}_A \neg F$	All A steps reach F.
`EF_A(`A`,`F`)`	$\mathsf{EF}_A F$	$\mathsf{E}[\mathsf{tt}\ {}_A\mathsf{U}\ F]$	Some A path reaches F.
`EF(`F`)`	$\mathsf{EF}F$	$\mathsf{EF}_{\mathsf{tt}} F$	Some path reaches F.
`AG_A(`A`,`F`)`	$\mathsf{AG}_A F$	$\neg\mathsf{E}[\mathsf{tt}\ {}_A\mathsf{U}\ \neg F]$	All A paths stay in F.
`AG(`F`)`	$\mathsf{AG}F$	$\mathsf{AG}_{\mathsf{tt}} F$	All paths stay in F.

4.2 Generating diagnostics in XTL

The standard libraries provided with XTL evaluate temporal operators by computing their denotational semantics, i.e. a temporal formula produces the set of states that satisfy it. For example, the EX_A operator, whose semantics is

$$[\![\mathsf{EX}_A F]\!] = \{s \mid \exists s \xrightarrow{a} s' \,.\, a \in A \wedge s' \in [\![F]\!]\}$$

is defined in the standard XTL library as

```
def EX_A (A : labelset, F : stateset) : stateset =
    { S : state where
        exists T : edge among out(S) in
            (label(T) among A) and (target(T) among F)
        end_exists
    }
end_def
```

This approach gives a linear complexity w.r.t. the size of the formula, but provides no justification of why the computed states satisfy the formula. For example, when some state s satisfies $\mathsf{EX}_A F$, we would like to exhibit a transition $s \xrightarrow{a} s'$ where a is in A and s' satisfies F. More generally, we seek a more sophisticated evaluation method that produces *explanations* [Ras91]: the evaluation of a temporal formula F on a state s

- evaluates whether F holds on s, and
- prints out a trace from s that confirms the result whenever possible.

Apparently this turns formulas into predicates over states, and thus temporal operators into functions over predicates, which would require a higher-order language. Nevertheless, a similar effect can be achieved thanks to the availability of macros in XTL. We turn formulas into open boolean expressions with a free variable `CURRENT` containing the current state, and operators into macros. For example, EX_A becomes

```
macro EX_A (A, F) =
    if      exists T : edge among out (CURRENT) in
                (label(T) among A) and
                (let CURRENT : state = target(T) in (F) end_let)
            end_exists
    then    do(print(T), true)
    else    false
    end_if
end_macro
```

Assuming `CURRENT` contains some state s, `T` ranges over transitions $s \xrightarrow{a} s'$ such that a is in `A` and s' satisfies `F`. The `let` construct binds (a fresh incarnation of) `CURRENT` to s') before evaluating `F`. The use of macros is essential: a function would evaluate `F` in the calling context instead, losing the opportunity to re-bind `CURRENT`. `do(`a`,`x`)` is a macro call that performs action a then returns x.

The other basic macro implements $\mathsf{E}[F\ {}_A\mathsf{U}\ G]$. The XTL code uses general iteration operators and will not be detailed here. In summary, the macro `EU_A(F,A,G)` performs a breadth-first search from `CURRENT` for a state that satisfies `G` through edges that match `F` and `A`, and stores the search tree. When a successful state is reached, it follows and prints a path through the search tree from that state back to the start state.

Since the search is breadth first, a *shortest* path is reported. This is a very useful property in practice, because facilitates the interpretation of the diagnostic: shortest paths contain the minimal sequence of events leading to the obtained result, whereas arbitrary paths may go through events that are irrelevent to the result being diagnosed. Combining several breadth-first searches does not necessarily result in the shortest overall trace, though.

All other operators can be defined in terms of (a slight generalization of) these two macros. In particular, since we work on finite models, infinite traces always end in circuits, which can be found using the following macro:

```
macro LOOP_A (F,A) =
    let MARK : state = CURRENT in
        (F) and EX_A(A, EU_A(F, A, CURRENT = MARK))
    end_let
end_macro
```

The body of LOOP_A keeps the start state in MARK, then searches for a path of at least one A step that stays in F and comes back to MARK.

Some evaluations of temporal formulas have no useful diagnostic trace: for example, if $\mathsf{EX}_A F$ does not hold in s, it means that *all* traces fail to reach F through an A step, so *all* outgoing edges from s should belong to the diagnostic. Such diagnostics generate huge output with little information; we do not attempt to produce them. In particular, ACTL operators come in pairs, performing dual quantification over traces: E operators (i.e. EX and E[_ U _]) search for the *existence of one path* satisfying certain criteria. Accordingly, the implementation of these operators will report that path as a diagnostic when the result is positive, but remain silent on negative results. Conversely, A operators will provide a diagnostic only on negative results. Nested E (or A) operators will produce consecutive pieces of a larger path. Conversely, an A nested inside an E should and will remain silent in all cases. A general discussion this issue can be found in [Ras91].

All diagnostic traces are produced "on the fly", that is, as soon as the XTL operator obtains the corresponding result. Because of this, the trace has to be printed backwards, since more deeply nested operators, which produce further parts of the trace, necessarily obtain their result first. Doing otherwise would require to store the trace and process it afterhand, which is currently not possible: XTL only supports *sets* (vs. *lists*) of edges, which are not adequate in to store traces (which may contain loops in all generality). This also restricts diagnostics to a single trace: for example, producing two diagnostic traces for conjunctive formulas $F \wedge G$ (one for F and one for G) would require to store the first one until a second one is obtained.

The generation of diagnostics adds a cost, in both terms of memory and time: since all operators are macros, formulas expand into big XTL expressions, which can stretch the XTL compiler to its limits. The linear complexity is also lost: n nested EU_A will produce n nested breadth-first searches, with an exponential worst-case complexity $O(k^n)$, where k is the size of the model. This is the price to pay to obtain diagnostic traces within the current XTL implementation. Things can be improved by pre-computing sub-formulas that will never be traced, although this requires a finer analysis from the user.

On the other hand, because everything gets expanded into a single expression, bound variables can be used in nested sub-formulas. This allows to capture and refer to intermediate states in the exploration, as illustrated in the LOOP_A macro above, and even more interestingly to propagate attribute values, allowing to express things such as “any message sent is eventually received”. The verification of CFS below shows the usefulness of this possibility.

4.3 Properties of CFS

Besides generic properties such as absence of deadlocks and non-determinism, our verification work focuses on the read/write coherency properties of the CFS protocol. These properties are expressed at the data level, in terms of events on gates `read` and `write`.

We proceed in two steps. First the formulas are evaluated on the `Abstract` model, where only the concerned gates are visible. The evaluation is fast since this model is very small. However the diagnostic traces are not informative because all the inner workings of the protocol have been abstracted. The interesting formulas are then evaluated again, this time on the `Complete` model. This takes longer but provides fully detailed diagnostic traces.

Since there is no CFS call for ending a read session, it was expected that the read/write coherency is rather loose. This coherency was not formally expressed at the start of this study, though, so the work consisted as much in determining the expected properties as in verifying them.

The following nine properties have been expressed and evaluated (the outcome of the evaluation is shown in parentheses):

1. *Global Liveness*: there is no global deadlock (holds).

```
AG(
  out(CURRENT) <> empty
)
```

2. *Determinism*: no state has non-deterministic transitions (fails).

```
AG(
  not(
    exists
       T1 : edge among out(CURRENT), T2 : edge among out(CURRENT)
    where
       (T1 <> T2) and (label(T1) = label(T2))
    end_exists
) )
```

3. *Local liveness*: at any time, all sites can eventually read and write (holds).

```
AG(
  EF(Dia(READ !SITE1 _, true)) and
  EF(Dia(READ !SITE2 _, true)) and
  EF(Dia(READ !SITE3 _, true)) and
  EF(Dia(WRITE !SITE1 _, true)) and
  EF(Dia(WRITE !SITE2 _, true)) and
  EF(Dia(WRITE !SITE3 _, true))
)
```

4. *Atomic coherency*: if no write occurs inbetween,

   4a. two different sites always read the same value (fails).

```
AG(
  Box(READ ?S1:Site ?V1:Val,
    AG_A(not(WRITE _ _),
      Box(READ ?S2:Site ?V2:Val where S1<>S2, V1=V2)
) ) )
```

   4b. a single site always reads the same value (fails).

```
AG(
  Box(READ ?S1:Site ?V1:Val,
    AG_A(not(WRITE _ _),
      Box(READ ?S2:Site ?V2:Val where S1=S2, V1=V2)
) ) )
```

   4c. if one site writes a value, another site will always read that value afterwards (fails).

```
AG(
  Box(WRITE ?S1:Site ?V1:Val,
    AG_A(not(WRITE _ _),
      Box(READ ?S2:Site ?V2:Val where S1<>S2, V1=V2)
) ) )
```

   4d. if one site writes a value, the same site will always read that value afterwards (holds).

```
AG(
  Box(WRITE ?S1:Site ?V1:Val,
    AG_A(not(WRITE _ _),
      Box(READ ?S2:Site ?V2:Val where S1=S2, V1=V2)
) ) )
```

5. *Propagation of values*: assuming a fair execution, if one site writes a value and no one writes inbetween, all sites will eventually read that value afterwards (holds).

```
AG(
  Box(WRITE ?S1:Site ?V1:Val,
    AG_A(not(WRITE _ _),
       EF_A(not(WRITE _ _), Dia(READ !SITE1 !V1, true)) and
       EF_A(not(WRITE _ _), Dia(READ !SITE2 !V1, true)) and
       EF_A(not(WRITE _ _), Dia(READ !SITE3 !V1, true))
) ) )
```

According to [QS83], we express the fairness assumption using the $\mathsf{AG}_A\mathsf{EF}_A F$ combination, which means: *it is always possible to eventually reach F while staying on A paths*, or yet, *all fair executions of A paths will eventually reach F*. Without the fairness assumption, we would have written $\mathsf{AF}_A F$ instead ($= A[\mathsf{tt}\ {}_A\mathsf{U}\ F]$).

6. *Sequential consistency*: if one site writes a value, no one writes inbetween, and another site reads that value, it cannot read another value afterwards (holds).

```
AG(
  Box(WRITE ?S1:Site ?V1:Val,
    AG_A(not(WRITE _ _),
      Box(READ ?S2:Site !V1,
        AG_A(not(WRITE _ _),
          Box(READ !S2 ?V2:Val, V1=V2)
) ) ) ) )
```

Compilation and evaluation of all nine formulas on the `Abstract` model takes 93 seconds of CPU time (on a Sun Ultra-1). As expected, evaluation on the `Complete` model takes much longer: 273 seconds and 52490 seconds (more than 14 hours) for properties 2 and 4b to report diagnostic traces of length 32 and 21, respectively. The trace produced for formula 4b follows, as an illustration. The transitions $s \xrightarrow{a} s'$ are displayed as triples (s, a, s'), where s and s' are given as internal state numbers.

```
  20  :             (4580, "READ !SITE2 !VAL2", 4580)
Box(A, F) is FALSE
  19  :          (474, "CFSANS !SITE2 !BEGINWRITE", 4580)
  18  :          (45433, "RCV !SITE2 !WRITEOK !SITE2", 474)
  17  :          (296, "SEND !SITE1 !WRITEOK !SITE2", 45433)
  16  :          (48817, "RCV !SITE1 !WRITERQ !SITE2", 296)
  15  :          (49464, "SEND !SITE2 !WRITERQ !SITE2", 48817)
  14  :          (33775, "CFSANS !SITE1 !ENDWRITE", 49464)
  13  :          (64972, "CFSREQ !SITE2 !BEGINWRITE", 33775)
  12  :          (46594, "RCV !SITE2 !INVALIDATE !SITE2", 64972)
  11  :          (47225, "CFSREQ !SITE1 !ENDWRITE", 46594)
  10  :          (900, "SEND !SITE1 !INVALIDATE !SITE2", 47225)
AG_A(A, F) is FALSE
  9   :       (900, "READ !SITE2 !VAL1", 900)
Box(A, F) is FALSE
  8   :    (884, "WRITE !SITE1 !VAL2", 900)
  7   :    (48772, "CFSANS !SITE1 !BEGINWRITE", 884)
  6   :    (37290, "CFSANS !SITE2 !READ", 48772)
  5   :    (49238, "CFSREQ !SITE1 !BEGINWRITE", 37290)
  4   :    (369, "RCV !SITE2 !READOK !SITE2", 49238)
  3   :    (308, "SEND !SITE1 !READOK !SITE2", 369)
  2   :    (172, "RCV !SITE1 !READRQ !SITE2", 308)
  1   :    (24, "SEND !SITE2 !READRQ !SITE2", 172)
  0   :    (0, "CFSREQ !SITE2 !READ", 24)
AG(F) is FALSE
*Failure.*
```

We use the possibility to bind and use attribute values to express properties 4a to 4d, 5 and 6 in their full generality, each as a single formula. For example, in property 4a, the variables `S1`, `V1`, `S2` and `V2` are used to capture and compare the sites and values of two consecutive `READ` events. Without this possibility, we would have had to repeat the formula for all possible combinations of values of these variables.

Let us now comment some evaluation results:

- Non-determinism is not a failure of the protocol but rather a consequence of the abstraction level of its LOTOS specification: diagnostic traces for property 2 show that the non-deterministic transitions correspond to possible reception of the same message from two different sources, a perfectly legal situation in the protocol.
- Failure of properties 4a, 4b and 4c is not surprising, since this kind of atomic coherency is quite strong and the protocol was known to have loose read synchronization mechanisms.
- Property 6 is more characteristic of typical distributed memory systems. It is related to the notion of *sequential consistency* [Mos93], which requires that events in the system are seen in the same order on all sites, though the time scales can be stretched or shifted from one site to another. In essence, property 6 captures the fact that values are read in the order in which they are written. This is even stronger than sequential consistency, which would allow two unrelated writes on different sites to be seen (at all sites) in the opposite order than that in which they really occurred.

If several memory blocks are considered, however, sequential coherency does no longer hold. This is illustrated on model `Abstract2` using the following property:

7. If one site writes values in two blocks in some order (and no other write occurs inbetween), another site cannot read the written value in the second block then fail to read the written value in the first block afterwards.

```
AG(
  Box(WRITE ?A1:Addr ?S1:Site ?V1:Val,
    AG_A(not(WRITE _ _),
      Box(WRITE ?A2:Addr !S1 ?V2:Val,
        AG_A(not(WRITE _ _),
          Box(READ !A2 ?S2:Site !V2,
            AG_A(not(WRITE _ _),
              Box(READ !A1 !S2 ?V3:Val, V1=V3)
) ) ) ) ) ) )
```

Property 7 is implied by sequential consistency and is not satisfied by `Abstract2`: intuitively, writes to different blocks can propagate at different speeds. CFS is thus sequentially consistent at the block level but not at the file level.

5 Conclusions

Although the LOTOS specification of CFS has a modest size, it produces excessively large models if processed in a straightforward manner. The modelling and verification work presented here illustrates how sophisticated tools can be used to achieve significant results in such complex cases.

Strictly speaking, the restrictions and simplifications needed to obtain a model of tractable size restrict the generality of the results we obtain. In this sense, model checking can be considered as a (very powerful) debugger, that is, a tool to find problems rather than prove their absence. In practice though, is is reasonable to expect most properties to be stable w.r.t the size of the system.

Beyond the validation of the CFS protocol, this case study has also demonstrated the practical applicability and usefulness of two technologies for dealing with complex specifications and supported by the CADP toolset:

- Using ALDÉBARAN and OPEN/CÆSAR technology to generate and compose models of different components of the system, we have been able to build up a model for a complete system that would have been impossible to produce in a single LOTOS compilation.
- CFS properties have been expressed and evaluated using the XTL temporal logic checker, with two important advantages. Firstly, XTL supports data, so that properties about values exchanged in the system have been conveniently expressed. Secondly, the XTL language is extensible, so that we have been able to define new temporal operators that produce execution traces to illustrate their results.

[Mat98] provides all the technical basis for the next version of XTL which will, hopefully, allow properties to be expressed even more conveniently and evaluated even more efficiently.

6 Acknowledgements

This study has been carried in close collaboration with both ARIAS/CFS developers and CADP tool designers. In particular, we would like to mention Thierry Jacquin and Daniel Hagimont on the former side, Hubert Garavel and Radu Mateescu on the latter. Many thanks too to some of them for their judicious comments on this text. Norman Ramsey's NOWEB literate programming system has been used to write the LOTOS specification. AIX is a trademark of IBM Corporation.

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.

[CGM+96] Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450. IFIP, Chapman & Hall, October 1996. Full version available as INRIA Research Report RR-2958.

[DHMdP96] Pascal Dechamboux, Daniel Hagimont, Jacques Mossière, and Xavier Rousset de Pina. The Arias Distributed Shared Memory: an Overview. In *23rd Intl Winter School on Current Trends in Theory and Practice of Informatics*, volume 1175 of *Lecture Notes in Computer Science*, 1996.

[dMRV92] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to Algebraic Specifications Based on the Language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[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.

[Fer89] Jean-Claude Fernandez. ALDEBARAN: A Tool for Verification of Communicating Processes. Rapport SPECTRE C14, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, September 1989.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.

[FKM93] Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. Symbolic Equivalence Checking. In C. Courcoubetis, editor, *Proceedings of the 5th Workshop on Computer-Aided Verification (Heraklion, Greece)*, volume 697 of *Lecture Notes in Computer Science*. Springer Verlag, June 1993.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[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.

[Gar98] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, Lecture Notes in Computer Science, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[GJM+97] Hubert Garavel, Mark Jorgensen, Radu Mateescu, Charles Pecheur, Mihaela Sighireanu, and Bruno Vivien. CADP'97 – Status, Applications and Perspectives. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, June 1997.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[Gut77] J. Guttag. Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[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.

[ISO89] ISO/IEC. LOTOS Description of the Session Protocol. Technical Report 9572, International Organization for Standardization — Open Systems Interconnection, Genève, 1989.

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

[KM97] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from Lotos Programs. In Ed Brinksma, editor, *Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands)*, volume 1217 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer Verlag. Extended version with proofs available as Research Report VERIMAG RR97-01.

[L94́] Luc Léonard. *The LOTOS Specification of the Enhanced Transport Service.* In *The OSI95 Transport Service with Multimedia Support*, pages 239–244 and 398–515. Springer Verlag, 1994.

[LBK+96] Guy Leduc, Olivier Bonaventure, Eckhart Koerner, Luc Léonard, Charles Pecheur, and Didier Zanetti. Specification and verification of a TTP protocol for the conditional access to services. In *Proceedings of 12th J. Cartier Workshop, Formal Methods and their Applications: Telecommunications, VLSI and Real-Time Computerized Control System*, Montreal, Canada, October 1996.

[Mat98] Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* Thèse de Doctorat, Institut National Polytechnique de Grenoble, April 1998.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mos93] David Mosberger. Memory Consistency Models. *Operating Systems Review*, 27(1):18–26, 1993.

[NV90] R. De Nicola and F. W. Vaandrager. Action versus State based Logics for Transition Systems. In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, 1990.

[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.

[Pec97] Charles Pecheur. Specification and Verification of the CO4 Distributed Knowledge System Using LOTOS. In Michael Lowry and Yves Ledru, editors, *Proceedings of the 12th IEEE International Conference on Automated Software Engineering ASE-97 (Incline Village, Nevada, USA)*, November 1997. Extended version available as INRIA Research Report RR-3259.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[Ras91] Anne Rasse. Error diagnosis in finite state systems. In *Proceedings of CAV'91*, number 575 in Lecture Notes in Computer Science. Springer Verlag, 1991.

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

Specification of the CFS coherency protocol in LOTOS

A 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 block of a CFS file (a *zone* in CFS terminology), 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]:

26a ⟨*sample* 26a⟩≡

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

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

26b ⟨*other sample* 26b⟩≡

```
(* ... *)
⟨sample 26a⟩
(* ... *)
```

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

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.

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

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).

B 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 block (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 block 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. CÆSAR's inefficiency in state representation is eliminated in subsequent minimizations.

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

C Data Types

C.1 Base Domains

Booleans are used throughout.

27a ⟨*data types* 27a⟩≡

```
library Boolean endlib
```

Defines:
`Bool`, used in chunk 29.

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.

27b ⟨*data types* 27a⟩+≡

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

Defines:
`Site`, used in chunks 30, 33–45, and 52.

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

28a ⟨*data types* 27a⟩+≡

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

Defines:
`Val`, used in chunks 30c, 31, 39b, and 45.

C.2 Interaction Primitives

`CfsCall` describes the CFS primitives offered to applications.

28b ⟨*data types* 27a⟩+≡

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

Defines:
`CfsCall`, never used.

`Message` defines the message exchanged between CFS entities.

28c ⟨*data types* 27a⟩+≡

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

Defines:
`Message`, used in chunks 30b and 39.

`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).

29 ⟨*data types* 27a⟩+≡

```
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 32.
Uses `Bool` 27a.

C.3 State Variables

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

30a ⟨*data types* 27a⟩+≡

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

Defines:
`SiteSet`, used in chunks 31 and 32.
Uses `Site` 27b 32.

`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.

30b ⟨*data types* 27a⟩+≡

```
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 31 and 32.
Uses `Message` 28c and `Site` 27b 32.

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

30c ⟨*data types* 27a⟩+≡

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

Defines:
`ValArray`, used in chunks 31 and 39b.
Uses `Site` 27b 32 and `Val` 28a.

Some complementary constants for convenience.

31 ⟨*data types* 27a⟩+≡

```
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` 30b, `SiteSet` 30a, `Val` 28a, and `ValArray` 30c.

D System Processes

D.1 CFS entity

The process `Site` describes the management of a single block by a CFS site. This is a state-oriented specification, originally based on the state machine presented in [Fas96]. All state is specified as process parameters:

- `state:State` encodes the control part of the state.
- `copies:SiteSet` holds the set of other sites that have obtained a copy of the block from this master site.
- `rqs:PktList` holds the list of pending requests from other sites received while this site is writing.

As a special case, the first site to request (read or write) access to the block 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`.

32 ⟨*processes* 32⟩≡

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

  ( ⟨local read 34a⟩ )

  []

  ( ⟨local beginwrite 34b⟩ )

  []

  ( ⟨local endwrite 35a⟩ )

  []

  ( ⟨remote readrq 35b⟩ )

  []

  ( ⟨remote writerq 35c⟩ )

  []

  ( ⟨remote readok 36a⟩ )

  []

  ( ⟨remote writeok 36b⟩ )

  []

  ( ⟨remote invalidate 36c⟩ )

  []

  ( ⟨transient flushrqs 37c⟩ )

  []
```

```
    ( ⟨transient forwardrqs 38⟩ )

    []

    ( ⟨transient invalwriting 37a⟩ )

    []

    ( ⟨transient invalinvalid 37b⟩ )

  endproc
```

Defines:
`Site`, used in chunks 30, 33–45, and 52.
Uses `PktList` 30b, `SiteSet` 30a, and `State` 29.

`InitSite` defines a site in initial state, i.e. with no valid copy of the block and no mastership.

33a ⟨*processes* 32⟩+≡

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

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

  endproc
```

Defines:
`InitSite`, used in chunk 46.
Uses `Site` 27b 32.

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

33b ⟨*processes* 32⟩+≡

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

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

  endproc
```

Defines:
`InitMaster`, used in chunk 46.
Uses `Site` 27b 32.

D.1.1 Local Requests

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

local read

34a ⟨*local read* 34a⟩≡

```
[(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` 27b 32.

local beginwrite

34b ⟨*local beginwrite* 34b⟩≡

```
[(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` 27b 32.

local endwrite

35a ⟨*local endwrite* 35a⟩≡

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

Uses `Site` 27b 32.

D.1.2 Remote Messages

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

remote readrq

35b ⟨*remote readrq* 35b⟩≡

```
  [(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` 27b 32.

remote writerq

35c ⟨*remote writerq* 35c⟩≡

```
  [(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` 27b 32.

remote readok

36a ⟨*remote readok* 36a⟩≡
```
  [state eq waitread] ->
  rcv !s !readok !s;
  cfsans !s !read;
  Site [cfsreq,cfsans,send,rcv] (s,valid,copies,rqs)
```
Uses `Site` 27b 32.

remote writeok

36b ⟨*remote writeok* 36b⟩≡
```
  [state eq waitwrite] ->
  rcv !s !writeok !s;
  cfsans !s !beginwrite;
  Site [cfsreq,cfsans,send,rcv] (s,writing,copies,rqs)
```
Uses `Site` 27b 32.

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

36c ⟨*remote invalidate* 36c⟩≡
```
  [ (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` 27b 32.

D.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`.

37a ⟨*transient invalwriting* 37a⟩≡
```
[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` 27b 32.

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

37b ⟨*transient invalinvalid* 37b⟩≡
```
[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` 27b 32.

transient flushrqs Answer the pending requests in `rqs`.

37c ⟨*transient flushrqs* 37c⟩≡
```
[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` 27b 32.

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

38 ⟨*transient forwardrqs* 38⟩≡

```
  [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` 27b 32.

D.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.

39a ⟨*processes* 32⟩+≡

```
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 49 and 52.
Uses `Message` 28c and `Site` 27b 32.

D.3 Memory

`Memory` holds the data (of sort `Val`) of the block 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
```

39b ⟨*processes* 32⟩+≡

```
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 52.
  `Memory`, never used.
Uses `Message` 28c, `Site` 27b 32, `Val` 28a, and `ValArray` 30c.

E Environment processes

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

E.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.

40 ⟨*processes* 32⟩+≡

```
  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 41.
`SlaveSiteProxy`, used in chunk 41.
Uses `Site` 27b 32.

To constitute an environment for a given site, `Site2Proxy` and `Site3Proxy` combine a single `MasterSiteProxy` with one `SlaveSiteProxy` for one and two other sites, respectively. 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.

41 ⟨*processes* 32⟩+≡

```
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 47.
  `Site3Proxy`, used in chunk 47.
Uses `MasterSiteProxy` 40, `Site` 27b 32, and `SlaveSiteProxy` 40.

E.2 Environment for Channels

`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.

42 ⟨*processes* 32⟩+≡

```
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 44.
`SlaveSendProxy`, used in chunk 44.
Uses `Site` 27b 32.

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

43 ⟨*processes* 32⟩+≡

```
  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 44.
Uses `Site` 27b 32.

`Channel2Proxy` and `Channel3Proxy` combine channel proxies to constrain a given channel, according to the expected number of sites (two and three, respectively). With the same reasoning as for site proxies, we can safely omit communications from a site to itself.

44 ⟨*processes* 32⟩+≡

```
  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
```

Defines:
`Channel2Proxy`, used in chunk 50.
`Channel3Proxy`, used in chunk 50.
Uses `MasterSendProxy` 42, `RcvProxy` 43, `Site` 27b 32, and `SlaveSendProxy` 42.

E.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 block any number of times;
- call `beginwrite` and `endwrite` before and after writing and/or reading the block any number of times.

45 ⟨*processes* 32⟩+≡

```
  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 chunks 51b and 52.
`ReadingUser`, never used.
`WritingUser`, never used.
Uses `Site` 27b 32 and `Val` 28a.

F 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

46 ⟨*processes* 32⟩+≡

```
  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
```

Defines:
`Master1`, used in chunk 48.
`Site1`, used in chunk 48.
`Site12`, never used.
`Site123`, never used.
`Site2`, used in chunk 48.
`Site3`, used in chunk 48.
Uses `InitMaster` 33b and `InitSite` 33a.

Proxy instances

47 ⟨*processes* 32⟩+≡

```
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
```

Defines:
`Proxy12`, used in chunk 48.
`Proxy123`, used in chunk 48.
`Proxy21`, used in chunk 48.
`Proxy213`, used in chunk 48.
`Proxy312`, used in chunk 48.
Uses `Site2Proxy` 41 and `Site3Proxy` 41.

Site instances with proxies

48 ⟨*processes* 32⟩+≡

```
  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
```

Defines:
`Master1With2`, never used.
`Master1With23`, never used.
`Site1With2`, never used.
`Site1With23`, never used.
`Site2With1`, never used.
`Site2With13`, never used.
`Site3With12`, never used.

Uses `Master1` 46, `Proxy12` 47, `Proxy123` 47, `Proxy21` 47, `Proxy213` 47, `Proxy312` 47, `Site1` 46, `Site2` 46, and `Site3` 46.

Cell instances

49 ⟨*processes* 32⟩+≡

```
  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
```

Defines:
`OutputCell1`, used in chunk 51a.
`OutputCell12`, never used.
`OutputCell123`, never used.
`OutputCell2`, used in chunk 51a.
`OutputCell3`, used in chunk 51a.
Uses `OutputCell` 39a.

Channel proxy instances

50 ⟨*processes* 32⟩+≡

```
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
```

Defines:
`ChannelProxy12`, used in chunk 51a.
`ChannelProxy123`, used in chunk 51a.
`ChannelProxy21`, used in chunk 51a.
`ChannelProxy213`, used in chunk 51a.
`ChannelProxy312`, used in chunk 51a.
Uses `Channel2Proxy` 44 and `Channel3Proxy` 44.

Cell instances with proxies

51a ⟨*processes* 32⟩+≡

```
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
```

Defines:
`OutputCell1with2`, never used.
`OutputCell1with23`, never used.
`OutputCell2with1`, never used.
`OutputCell2with13`, never used.
`OutputCell3with12`, never used.

Uses `ChannelProxy12` 50, `ChannelProxy123` 50, `ChannelProxy21` 50, `ChannelProxy213` 50, `ChannelProxy312` 50, `OutputCell1` 49, `OutputCell2` 49, and `OutputCell3` 49.

General User instances

51b ⟨*processes* 32⟩+≡

```
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
```

Defines:
`GeneralUser1`, never used.
`GeneralUser2`, never used.
`GeneralUser3`, never used.
Uses `GeneralUser` 45.

G 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 block by three concurrent sites. An initial `firstmaster` message is generated spontaneously before the channel starts its normal operation.

52 ⟨*behaviour* 52⟩≡

```
  (
    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` 45, `InitMemory` 39b, `OutputCell` 39a, and `Site` 27b 32.

Finally, here is the specification itself.

53 ⟨*cfs.LOTOS* 53⟩≡

```
(***************************************************************
  Compiled from @(#)cfs.nw      4.5 - 98/04/24
  Charles Pecheur, INRIA Rhone-Alpes
***************************************************************)

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

  ⟨data types 27a⟩

behaviour

  ⟨behaviour 52⟩

where

  ⟨processes 32⟩

endspec
```

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

Index of LOTOS Identifiers

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399