Advanced Modelling and Verification Techniques Applied to a Cluster File System

Charles Pecheur
RIACS / NASA Ames Research Center
MS 269-2, Moffett Field, CA 94035, USA

E-mail: pecheur@ptolemy.arc.nasa.gov

Abstract

This paper describes the application of advanced formal modelling techniques and tools from the CADP *toolset to the verification of* CFS*, a distributed file system kernel. After a short overview of the specification of* CFS*, we describe the techniques used for model generation and verification, 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 verify properties of the system. In particular, an* XTL *extension providing richer diagnostics is presented.*

Note This research was performed at and for INRIA Rhône-Alpes, France, and is not connected to any RIACS and NASA activity.

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 paper illustrates the use of advanced techniques for the modelling and verification of CFS (Cluster File System) [5], a distributed file system built on top of the ARIAS shared memory architecture [4]. Two original aspects are put forth:

- the use of *compositional model generation* [8], 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 [20] and the development of an XTL extension providing richer diagnostics.

The rest of this first section gives a survey of the LOTOS specification language and tools 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 The LOTOS language and tools

The specification of CFS has been written in LOTOS, a Formal Description Technique for the description and analysis of complex communication protocols and distributed systems. LOTOS is a language standardized by ISO [15], with a high abstraction level and a strong mathematical basis. It has been applied to many complex systems such as network services and protocols [16, 19] but also cryptographic protocols [18] or hardware architectures [2].

LOTOS consists of two "orthogonal" sub-languages: a *data part* based on algebraic abstract data types and derived from the ACT ONE specification language [3], and a *control part* based on a process algebra, combining the best features of CCS [21] and CSP [14]. A system is modelled as a collection of concurrent processes interacting by synchronous rendez-vous, and those processes can manipulate data values and exchange them through their interactions. This paper does not show any LOTOS source code, so no familiarity with the language is necessary. The interested reader can find tutorials for LOTOS in the literature, e.g. [1, 27].

The model (i.e. the meaning) of a LOTOS specification is defined as the graph of all its possible actions (a *Labelled Transition System*, or LTS). These graphs can be minimized and compared according to different equivalence and refinement criteria. In this study, we use observational equivalence [21] for minimization, that is, we reduce models into minimal observationally equivalent ones.

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. All the work reported here has been done within the framework of the EUCALYPTUS LOTOS Toolset [10], an X-Windows based, user-friendly interface federating several complementary LOTOS tools from different sources. An important part of EUCALYPTUS is the CÆSAR/ALDEBARAN Development Package (CADP) [7, 12], a leading edge toolbox dedicated to the formal verification 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. The following CADP tools were mainly used in this application:

- CÆSAR [13] compiles a LOTOS program into its transition graph. The data part is translated into executable C code, which is used to compute the graph [9].
- ALDEBARAN [6] is a verification tool for comparing or minimizing graphs with respect to any of several simulation and bisimulation relations.
- XTL [20] 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.

CADP also provides OPEN/CÆSAR [11], an open programming interface that separates the tasks of graph generation and graph exploration, allowing different applications (simulation, controlled execution, model checking) to be applied to different formalisms (LOTOS programs, explicit graphs, networks of automata). It has been used here to compose graphs of communicating processes together.

The APERO data type pre-processor [24] has also been used to define most data types in the specification. 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.). Besides reducing the size of data type declarations, these notations are also much more readable, avoid the burden of algebraic definitions and hide the technical complications needed to allow the compilation of algebraic data type definitions.

2 Specification of CFS

2.1 Presentation of ARIAS and CFS

ARIAS [4] is a shared memory support system implemented as an extension of IBM's 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) [5] 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.

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 [5]. 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 is 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.

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

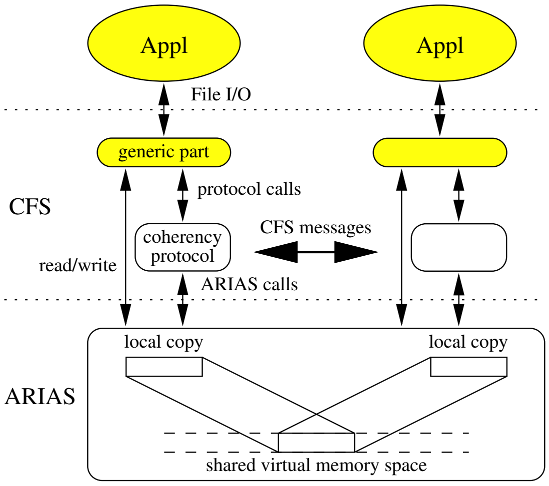

Figure 1. ARIAS and CFS

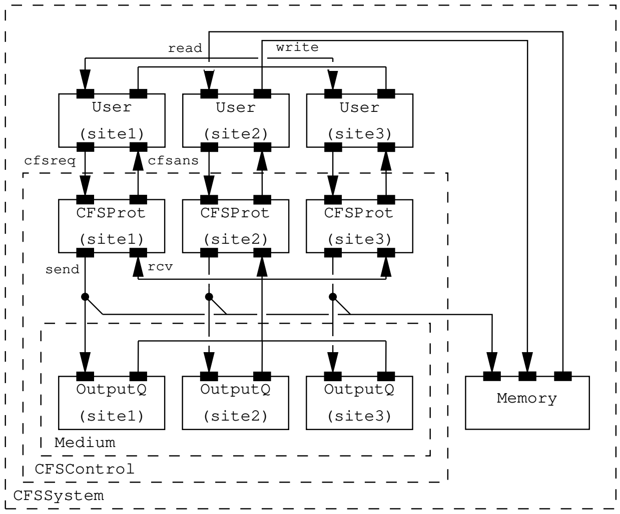

Figure 2. Structure of the LOTOS specification of CFS

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. 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).

The complete specification is about 1000 lines long, and is provided with an extended version of this article in [25]. Its structure is shown on Figure 2.

The CFS protocol is specified for a given site as a process `CFSProt` with four connections: `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. The process `Medium` models the transmission of these messages through the underlying ARIAS system. For modelling purposes, the buffering capacity is limited to one message per sending site: `Medium` is made of single-slot queues `OutputQ`, one per site. These processes together constitute the `CFSControl` process, that models correlation between CFS primitives for acquiring and releasing access to a CFS block.

The process `Memory` models the memory management part of ARIAS. It holds the local copies of the block at each site and applies `read` and `write` events on them. It also observes the CFS protocol messages and propagates master copies between sites accordingly. To complete the picture, a process `User` for each site correlates memory accesses and CFS calls, modelling the correct use of CFS synchronization primitives: for example, `write` events are only allowed between `beginwrite` and `endwrite` CFS calls.

2.3 The CFS Protocol

The source CFS definition [5] describes the protocol in terms of an input-output state automaton. For illustration, Figure 3 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.

The definition of process `CFSProt` essentially captures that automaton (several rounds of discussions with the designers were needed to reach the level of accuracy needed for formal specification). The state variables of the protocol become parameters of `CFSProt`. The body of `CFSProt` is a non-deterministic choice between reception of CFS requests and messages, with infinite looping modelled by `CFSProt` calling itself recursively.

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 `CFSControl` alone produced a (essentially un-

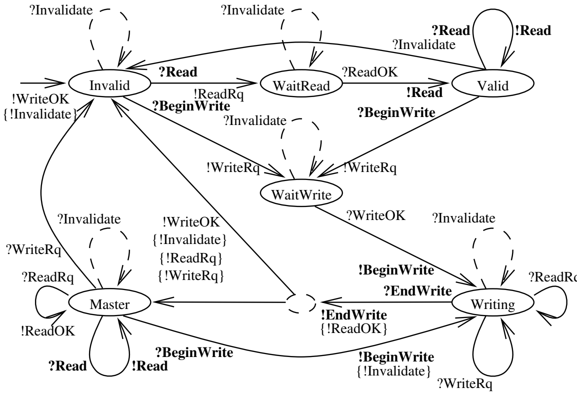

Figure 3. State of a block in the CFS protocol

usable) 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.

3.1 Tools for Compositional Model Generation

Compositional generation is provided by a tool EXP.GEN that takes a description of a network of communicating automata (expressed in LOTOS-like syntax), the graphs of these automata, and generates the graph for the full network[1]. Using this tool, compositional generation is obtained by:

- decomposing the whole system into parts of manageable size,
- generating the model for each part using CÆSAR, and minimizing it (modulo observational equivalence) using ALDEBARAN,
- combining those models using EXP.GEN into models for increasingly large parts, while minimizing after each step with ALDEBARAN, until obtaining the model of the whole system.

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 compromising 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 [17]. This is sound provided that the substitution does not modify the global model, i.e. $(P||E_P)||Q = P||Q$. 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.

3.2 Generation of a Model of CFS

From the specification depicted on Figure 2, the graph for `CFSSystem`, modelling the management of a single CFS block, has been generated compositionally, using environments to reduce the initial sizes of processes `User` and `OutputQ`. Table 1 gives the sizes of the different components and the generation and minimization times (in seconds, on a Sun Ultra-1 workstation). Two derived models have also been generated:

`Abstract` is `CFSSystem` 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. The minimization of `Abstract` takes more than 5 hours but results in a small, highly symmetrical model with only 14 states.

`Abstract2` consists of 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. Since this is a fully non-synchronized composition, we get the worst case of state space explosion[2]. According to this, combining two concurrent instances of `CFSSystem` in a similar way would go far beyond available computing resources.

4 Verification

4.1 Overview of XTL

The properties of the CFS protocol have been expressed and evaluated as temporal logic formulas, using the XTL tool. 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

[1] Technically, EXP.GEN uses OPEN/CÆSAR to combine EXP.OPEN, which computes the graph for the network of automata, and GENERATOR, which does the exhaustive graph generation.

[2] for n independent components, the number of states is $S_n = {S_1}^n$ and the number of transitions is $T_n = n.T_1.{S_1}^{n-1}$.

Table 1. Model generation statistics

process	gen.	min.	#states	#trans
Site || EnvSite ($\times 3$)	5.0	0.1	75	130
OutputQ || EnvOutputQ ($\times 3$)	4.5	<0.1	13	30
Medium	3.8	3.1	2,197	15,210
CFSControl	8.0	7.9	11,031	34,728
User ($\times 3$)	2.0	0.1	6	14
Memory	19.9	57.5	8	504
UserMemory	9.5	13.7	1,728	103,680
CFSSystem	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

manipulating them (e.g. initial state, incoming and outgoing transitions of a state, source and target states of a transition). It can also do pattern matching on the labels of transitions and thus access the individual attributes of structured actions. For example, the following XTL expression computes the set of all states that can take a transition on `G` with integer attribute larger than 10:

```
{ S : state where
    exists T : edge among out(S) 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 μ-calculus) are provided as XTL libraries. ACTL [23], a variant of CTL applicable to labelled transition systems, has been used here. The following four primitive ACTL operators are used, for any set of events A and formulas F, G:

XTL syntax	math syntax
EX_A(A,F)	$\mathsf{EX}_A F$
AX_A(A,F)	$\mathsf{AX}_A F$
EU_A(F,A,G)	$\mathsf{E}[F\ {}_A\mathsf{U}\ G]$
AU_A(F,A,G)	$\mathsf{A}[F\ {}_A\mathsf{U}\ G]$

Informally, $\mathsf{EX}_A F$ (resp. $\mathsf{AX}_A F$) means *some (resp. all) path reaches F through an A step*, and $\mathsf{E}[F\ {}_A\mathsf{U}\ G]$ (resp. $\mathsf{A}[F\ {}_A\mathsf{U}\ G]$) means *some (resp. all) path stays in F through A steps until it reaches G*. The following derived notations are also used:

XTL syntax	math syntax	definition
Box(A,F)	$[A]F$	$\neg\mathsf{EX}_A \neg F$
AG_A(A,F)	$\mathsf{AG}_A F$	$\neg\mathsf{E}[\mathsf{tt}\ {}_A\mathsf{U}\ \neg F]$
AG(F)	$\mathsf{AG}F$	$\mathsf{AG}_{\mathsf{tt}} F$

Intuitively, $[A]F$ means *F holds after all A*, and $\mathsf{AG}_A F$ means *F remains true through all A traces* (through all traces for $\mathsf{AG}F$).

4.2 Generating diagnostics in XTL

The standard libraries provided with XTL evaluate temporal formulas in a functional, bottom-up manner, by computing the set of states that satisfies each sub-formula. 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.

We have developed an alternative version of the ACTL library that produces *explanations*, as defined in [26]: 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 this makes sense.

The implementation relies on the availability of macros in XTL. Formulas are turned into open boolean expressions with a free variable `CURRENT` containing the current state, and temporal 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` is some state s, `T` ranges over transitions $s \xrightarrow{a} s'$. If there is a `T` such that a is in `A` and s' satisfies `F`, then it is printed as an explanation. The `let` construct binds (a fresh incarnation of) `CURRENT` to s' before evaluating `F`. `do(`a`,`x`)` performs action a then returns x. The use of macros is essential: a function would evaluate `F` in the calling context instead, losing the opportunity to re-bind `CURRENT`.

Another macro implements $\mathsf{E}[F\ {}_A\mathsf{U}\ G]$. 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. All other operators can be defined in terms of (a slight generalization of) these two macros.

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, 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 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 `CFSSystem` 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 by the CFS protocol designers, 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).
2. *Determinism*: no state has non-deterministic transitions (fails).
3. *Local liveness*: at any time, all sites can eventually read and write (holds).
4. *Atomic coherency*: if no write occurs inbetween,
   - 4a. two different sites always read the same value (fails).
   - 4b. a single site always reads the same value (fails).
   - 4c. if one site writes a value, another site will always read that value afterwards (fails).
   - 4d. if one site writes a value, the same site will always read that value afterwards (holds).
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).
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).

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 `CFSSystem` 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.

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, property 4a is defined as

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

where 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* [22], 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 consistency does no longer hold. This is illustrated on model `Abstract2` using the following property:

7. If one site writes values in two different 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.

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 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 more advanced techniques 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. Nevertheless, verifying properties on small simplified models still significantly improves the confidence that these properties hold in the general case.

Beyond the verification 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 ALDEBARAN 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 generation.

- 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.

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

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

[2] G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In R. Gotzhein and J. Bredereke, editors, *Proceedings of FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450. IFIP, Chapman & Hall, Oct. 1996.

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

[4] P. Dechamboux, D. Hagimont, J. Mossière, and X. R. 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.

[5] J.-P. Fassino. Utilisation d'une mémoire virtuelle répartie pour le support d'un système de fichiers réparti. DEA thesis, Université Joseph Fourier, Grenoble, June 1996.

[6] J.-C. Fernandez. ALDEBARAN: A Tool for Verification of Communicating Processes. Rapport SPECTRE C14, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, Sept. 1989.

[7] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In R. Alur and T. A. Henzinger, editors, *Proceedings of CAV'96 (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, Aug. 1996.

[8] J.-C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic Equivalence Checking. In C. Courcoubetis, editor, *Proceedings of CAV'93 (Heraklion, Greece)*, volume 697 of *Lecture Notes in Computer Science*. Springer Verlag, June 1993.

[9] H. Garavel. Compilation of LOTOS Abstract Data Types. In S. T. Vuong, editor, *Proceedings of FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, Dec. 1989.

[10] H. 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.

[11] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proceedings of TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, Mar. 1998. Springer Verlag.

[12] H. Garavel, M. Jorgensen, R. Mateescu, C. Pecheur, M. Sighireanu, and B. Vivien. CADP'97 – Status, Applications and Perspectives. In I. Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, June 1997.

[13] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of PSTV'90 (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

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

[15] 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, Sept. 1988.

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

[17] J.-P. Krimm and L. Mounier. Compositional State Space Generation from Lotos Programs. In E. Brinksma, editor, *Proceedings of TACAS'97 (University of Twente, Enschede, The Netherlands)*, volume 1217 of *Lecture Notes in Computer Science*, Berlin, Apr. 1997. Springer Verlag.

[18] G. Leduc, O. Bonaventure, E. Koerner, L. Léonard, C. Pecheur, and D. Zanetti. Specification and verification of a TTP protocol for the conditional access to services. In *Proceedings of 12th J. Cartier Workshop*, Montreal, Canada, Oct. 1996.

[19] L. 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.

[20] R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. Doctorate thesis, Institut National Polytechnique de Grenoble, Apr. 1998.

[21] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

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

[23] R. D. 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.

[24] C. Pecheur. *Improving the Specification of Data Types in LOTOS*. Doctorate thesis, University of Liège, Nov. 1996. Faculty of Applied Sciences, Pub. Coll. Nr 171.

[25] C. Pecheur. Advanced Modelling and Verification Techniques Applied to a Cluster File System. Research Report RR-3416, INRIA, Grenoble, May 1998.

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

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