Specification and Verification of the Co4 Distributed Knowledge System Using LOTOS

Charles Pecheur
INRIA Rhône-Alpes
655 avenue de l'Europe
38330 Montbonnot Saint Martin, France
Charles.Pecheur@inria.fr

Abstract

This paper relates the formal specification and verification of a consensual decision protocol used in CO_4*, a computer environment dedicated to the building of a distributed knowledge base. This protocol has been specified in the* ISO *formal description technique* LOTOS*. The* CADP *tools from the* EUCALYPTUS LOTOS *toolset have been used to verify different safety and liveness properties. The verification work has confirmed an announced violation of knowledge consistency and has put forth a case of inconsistent hierarchy, four cases of unexpected message reception and some further local corrections in the definition of the protocol.*

1 Introduction

The need for formal verification in the design of complex distributed systems is now widely recognized. 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*, which is more general but requires human assistance for the proof, and *model checking*, which proceeds autonomously but is applicable only to systems with a finite state space.

This article relates a formal verification case study in the field of distributed knowledge bases. CO_4 is a computer environment dedicated to the incremental and concurrent building of a knowledge base [4]. In particular, communication between CO_4 entities follow a consensual decision protocol similar to scientific paper reviewing policies.

This protocol has been specified in the ISO formal description technique LOTOS [14], and the CADP (CÆSAR/ALDÉBARAN) toolset [9] has been used to verify different expected properties on this specification. These tools belong to the model checking family; an important part of the work has been to achieve a finite state space of manageable size while keeping a realistic description of the CO_4 protocol.

The article is organized as follows: section 2 gives a short introduction to the LOTOS language, section 3 describes the main features of a CO_4 system along with their LOTOS specification and section 4 presents the verification tools, methodology and results.

2 The LOTOS language

LOTOS [14] 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.

This paper does not assume familiarity with LOTOS from the reader, but due to the limited available space only a very short intuitive overview follows. This should be sufficient to understand the essence of the forthcoming excerpts. Tutorials for LOTOS are available, e.g. [1, 23].

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 [12], more specifically on the ACT ONE specification language [3]. Data types are defined using an equational formalism, which we will not present here. As a matter of fact, most data types of our specification are defined in a much more concise and readable extended syntax [21].

The control part is based on a process algebra, combining the best features of CCS [18] and CSP [13]. A concurrent system is described as a collection of processes interacting by rendezvous. The behaviour of each process is built compositionally using an algebra of operators. Behaviours can manipulate data values and exchange them through their interactions. The main operators of LOTOS are summarized in table 1.

LOTOS has been applied to many complex systems such as OSI services and protocols [15, 16] but also cryptographic protocols [17] or hardware components [2]. 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. The tools used in the

`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

work presented here are introduced in more details in section 4.

3 Presentation and Specification of Co4

This section describes the main features of CO_4, and explains how they have been specified in LOTOS. Each subsection is divided into a *presentation* part, which presents some aspect of CO_4, and a *specification* part, which explains how it is translated into LOTOS. We focus on architecture and general principles; we do not detail the messages exchanged in the CO_4 protocol. The reference document used for writing the specification is [5].

The LOTOS specification is intended for model-based verification using the CADP tools, as described in section 4. To take full advantage of these techniques, the specification must produce a *model* (the graph of all possible states) of finite and tractable size. This is taken into account when writing the specification: the behaviour part has a fixed finite (and not too large) number of concurrent processes, we avoid choices over wide ranges of data values, we consider only constrained execution scenarios rather than the full spectrum of all possible system behaviours. Of course, this restricts the generality of the results we will obtain, but in most cases small and large systems exhibit very similar properties.

Most data types have been defined using the syntax extensions provided by the APERO pre-processor [21]. These extensions provide convenient concise declarations for many common data structures such as records, enumerations, sets, lists, or even ML-style "algebraic" types [19]. 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 (331 vs. 1266 lines of data type definitions), these notations are also much more readable, avoid the burden of equational definitions and hide the technical complications due to theoretical limitations of the CADP data type compiler.

The complete specification (with APERO notations) is about 1400 lines long. The full document, in litterate programming style, is available as an annex of [22]; only small significant excerpts are given here, in which some technical details have been wiped out for the sake of clarity.

3.1 Top-Level Structure

Presentation A CO_4 system is made up of a collection of *(knowledge) bases*, or KB*s*, organized as a tree, as illustrated on figure 1. The leaf nodes are *individual* KB*s* directly connected to the external world, and holding the knowledge of a single human user (or a group thereof). Intermediate nodes are *group* KB*s*; their children are the *members* (or *subscribers*) of the group. Their content is the common knowledge consensually accepted by all their members. This structure is dynamic: new KBs may register themselves to group KBs while the system is running.

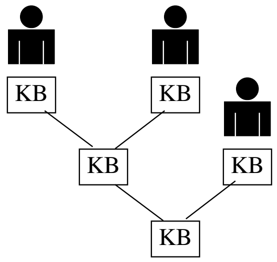

Figure 1. A Co4 hierarchy

The CO_4 protocol defines the messages exchanged between group KBs and their member (group or individual) KBs, in order to build their common knowledge bases.

Specification The core of the LOTOS specification is a pair of process definitions `IndividualKB` and `GroupKB`, modelling resp. a single individual or group KB. We use separate LOTOS gates to model incoming and outgoing interactions of each base, and we further separate communications with children vs. parent bases. This separation isolates unrelated interactions, as part of our efforts to limit the cost of the verification work. A group base thus has four gates: `parentin`, `parentout` to communicate with its parent base and `childrenin`, `childrenout` to communicate with its children. Individual bases only have `parentin` and `parentout` but also a gate `user` to interact with the external user. In addition, all bases have a gate `signal` that supports events used to monitor some internal situations. This is illustrated on figure 2.

It is possible to describe an unbounded pool of concurrent KBs in LOTOS, using recursive process definitions like

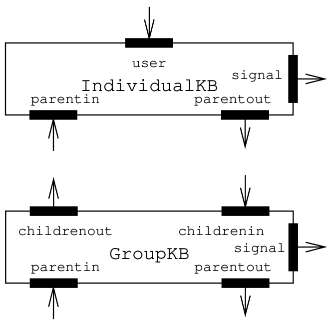

Figure 2. Interface of knowledge bases

`KBPool := KB ||| KBPool`. However, this is not suitable for model checking: the CADP compiler requires a fixed number of concurrent processes. We therefore define a static (and small) hierarchy of individual and group KBs.

Furthermore, even a single KB can store an unbounded amount of data and thus has infinitely many states. To obtain a finite and tractable state space, we further restrict the explored scenarios by coupling the system with a process `UserInput`, playing the role of the CO_4 system users and performing only a few specific user actions. The resulting top-level structure of the LOTOS specification is:

```
specification CO4System [user,signal] : noexit
  ... (* data type definitions *)
behaviour
  UserInput [user]
  |[user]|
  KBHierarchy [user,signal]
where
  ... (* process definitions *)
endspec
```

Different definitions of `UserInput` and `KBHierarchy` are used to analyze different scenarios. The `KBHierarchy` process is made up of concurrent KB processes, whose interconnection structure follows the intended hierarchy. For example, a hierarchy with two individual and one group base is shown on figure 3. The corresponding LOTOS process definition follows:

```
process KBHierarchy [user,signal] : noexit :=
  hide up1,down1,up2,down2 in (
    ( IndividualKB [user,down1,up1,signal]
                   (base1)
      |||
      IndividualKB [user,down1,up1,signal]
                   (base2) )
    |[up1,down1]|
    GroupKB [up1,down1,down2,up2,signal]
            (base3)
    |[up2,down2]|
    stop (* base3 has no parent *) )
endproc
```

3.2 Data Structures

Presentation CO_4 makes use of various kinds of simple and compound data structures, which we can sort out in four categories:

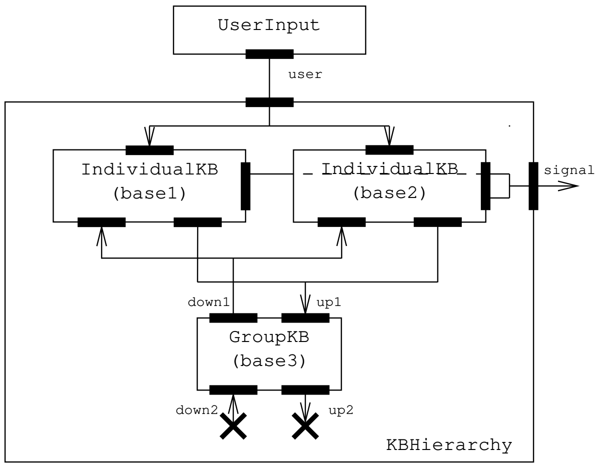

Figure 3. Top-level specification with 3 bases

- *Identifiers* such as *message identifiers* (surrogates in [5]) and *base identifiers* are just flat name spaces.
- *Messages* are exchanged according to the CO_4 protocol. There are ten different message primitives, with a well-defined structure for each one.
- *State variables* store the state information of each KB. The most complex ones are sets of tuples, which are indeed used as tables indexed by message identifiers.
- *Knowledge data* represent the knowledge accumulated in KBs. CO_4 knows of two kinds of knowledge-related data: a (knowledge) *repository* is a body of knowledge as accumulated in KBs, and a *proposal* is a (proposed) modification of a repository.

Knowledge is manipulated by CO_4, but the precise structure of this knowledge is transparent to the CO_4 protocol. [5] assumes that "concurrent proposals are independent", meaning that if each of them is applicable to a repository, then they can be applied together in any order. This assumption is necessary to ensure that the consistency of knowledge is preserved in every case.

Specification The first three kinds of data types present no particular difficulty and are straightforwardly specified in LOTOS using concise APERO syntax extensions. We only discuss the specification of knowledge data, which is deserves a little more attention.

We model a *repository* K as a set of *atoms* $\{a_1, \ldots, a_n\}$, and a *proposal* p as a sequence $(c_1, \ldots, c_n)$ of insertions and removals of atoms.

The assumption of independence has two possible interpretations:

(a) the *structure of knowledge* is such that *any* two proposals (and concurrent proposals in particular) are always independent,

(b) the *users of the system* make sure that *concurrent* proposals are always independent.

Both are not realistic: interpretation (a) leads to a very restrictive model of knowledge bases, whereas interpretation (b) requires "rational" users that can detect and avoid contradictory proposals. In our LOTOS specification we take interpretation (b), which leaves the possibility to study the behaviour of the system in the case of contradictory concurrent proposals. This also allows us to test how the protocol can be modified to preserve consistency in this case. We define this as a binary *compatibility relation* over atoms, such that a base is consistent if it does not contain incompatible atoms.

3.3 The Co4 Protocol

Presentation The principles underlying the Co$_4$ protocol are those of peer-reviewing: before being accepted, every request must be submitted and accepted by all members of the group. Typically, the processing of a new proposal takes five stages:

$s \text{ –achieve}(p)\!\rightarrow g$	a member s submits a proposal p to its group base g,
$s_1, \ldots, s_n \leftarrow\!\text{ask-all}(p)\text{– } g$	g broadcasts p among all its members $s_1, \ldots, s_n$,
$s_1, \ldots, s_n \text{ –reply}(p, \text{accept})\!\rightarrow g$	each s_k votes on p,
$s \leftarrow\!\text{notify}(p, \text{accept})\text{– } g$	g reports the result of the vote to the author s,
$s_1, \ldots, s_n \leftarrow\!\text{tell}(p)\text{– } g$	if p is accepted, g applies and publishes it to all members.

This policy is used for additions to the knowledge base, but also for the registration of new members and other transactions. In addition, any request can be cancelled by its author before it is accepted.

[5] describes each KB as an input-output automaton and defines the protocol in terms of transitions, in 35 rules of the following general form:

$$\frac{s \text{ –}m_1(x)\!\rightarrow b}{\begin{array}{c} V := y \\ b \text{ –}m_2(z)\!\rightarrow r \end{array}} \quad P$$

according to this rule, when base b receives a message m_1 with content x from base s such that condition P is true, it sets its state variable V to y and sends a message m_2 with content z to base r (P, y, z, r are expressions that depend on x and s). This is merely a general sketch; the smaller details vary from rule to rule. As a concrete example, the rule *handle-notify* is defined as

$$(\textit{handle-notify}) \qquad \frac{g \text{ –pool-notify}(n, \_)\!\rightarrow b}{P := P - \{\langle n, \_, \_\rangle\}} \quad \langle n, \_, \_\rangle \in P$$

that is, a base b receiving a pool-notify message from its parent g , related to a request n present in state variable P, removes the entry for n from P (the "_" denote "don't care" values).

Specification The LOTOS behaviour of a KB process is a loop, where one protocol rule is applied at each iteration, accessing and possibly modifying state variables. In LOTOS (as in other applicative languages), this is defined as a recursive process with the state variables as parameters. Each rule is modelled as a branch of a choice ending with a recursive call. Here is, for example, the LOTOS specification of the above *handle-notify* rule:

```
process IndividualKB [...]
     (b:BaseId, g:BaseId, ..., P:PendingTbl)
       : exit :=
...
parentin ?from : BaseId !b ?msg : Message;
( ... []
  [is_poolnotify(msg)] ->
  [g = from] ->
  ( let n : Id = inreplyto(msg) in
    [n isin P] ->
    IndividualKB [...] (b, g, ..., remove(n,P)) )
  [] ... )
endproc
```

4 Verification of Co4

Producing a formal specification of a system is a fruitful investment, even before any formal analysis of the meaning of that specification is attempted: it requires a thorough and systematic analysis of the source description of the specified system; be it laid down on paper or still inside the mind of its designer. All implicit facts, informal statements, language short cuts or misnomers have to be clarified. In particular, static type checking immediately spots many of these kinds of imprecisions.

This was clearly the case with the LOTOS specification of the Co$_4$ protocol. Though formal in style, the rule-based definition of the protocol in [5] is hand-made and completed by many informal remarks. A bunch of imprecisions were detected during the writing of the specification, such as a data structure showing different components at different places in the description, or improper identifiers associated to a message—this results in a typing error in the LOTOS specification.

In the following sub-sections, we describe the verification tools that we used, how they were applied, the properties that were considered and the results obtained from their formal verification.

4.1 LOTOS Tools

All the processing of LOTOS specifications has been done within the framework of the EUCALYPTUS LOTOS Toolset [9], an X-Windows based user-friendly interface that federates several complementary LOTOS tools from different sources. Besides the APERO data type pre-processor described in section 3, the EUCALYPTUS toolset also contains CADP (CÆSAR/ALDÉBARAN Development Package), a leading edge toolbox dedicated to the 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.

In our case study, we have used the following CADP tools:

- CÆSAR [11] and CÆSAR.ADT [8] are compilers that transform respectively the control and data part of a LOTOS program into a state graph describing its exhaustive behaviour (this kind of graph is called a *Labelled Transition System*, or LTS). This graph can be represented either *explicitly*, as a set of states and transitions, or *implicitly*, as a library of C functions allowing to execute the program behaviour in a controlled way.

- ALDÉBARAN [7] is a verification tool for comparing or minimizing graphs with respect to any of several simulation and bisimulation relations [20, 18]. Initially designed to deal with explicit graphs produced by CÆSAR, it has been extended to also handle networks of communicating automata (for on-the-fly and symbolic verification).

- XSIMULATOR is an interactive program for exploring the behaviour of a LOTOS specification. It allows back and forth step-by-step execution, to walk through the different alternative branches of the graph.

- EXHIBITOR performs a search in the graph, looking for execution sequences that start from the initial state and match a specified pattern. This pattern combines boolean operators and (a subset of) regular expressions, and allows to characterize deadlock states.

Both EXHIBITOR and XSIMULATOR accept graphs in either explicit or implicit forms, and therefore can be applied to graphs of untractable or even infinite size. Nonetheless, it is still crucial to reduce the state space as much as possible; in particular, if the searched trace cannot be found then EXHIBITOR will run forever.

4.2 Methodology and statistics

The LOTOS specification described in section 3 has been used to verify several properties of the CO_4 protocol, using the CADP toolset inside the EUCALYPTUS environment.

At an early stage, XSIMULATOR is used for a first exploration of the LOTOS specification. Some missing rules have been detected in CO_4 using simulation alone. When the most obvious problems have been fixed, we turn to more systematic exploration techniques.

For the majority of scenarios, the complete graph is generated with CÆSAR and minimized with ALDÉBARAN. We then apply EXHIBITOR to the obtained graph to search for traces leading to particular situations: deadlock states, knowledge inconsistencies, etc. In more complex scenarios, where the graph is too big to be completely generated, EXHIBITOR is applied to the LOTOS specification itself, using on-the-fly graph generation. XSIMULATOR is still used to re-play the sequences found and get a better understanding of what is going on.

We have considered different KB hierarchies and, for each hierarchy, different user environments. More general scenarios give more confidence in the validity of the results obtained, but also more complex specifications and models and therefore higher computation costs to obtain them, if at all possible. Four different hierarchies have been taken into consideration. They are depicted in figure 4.

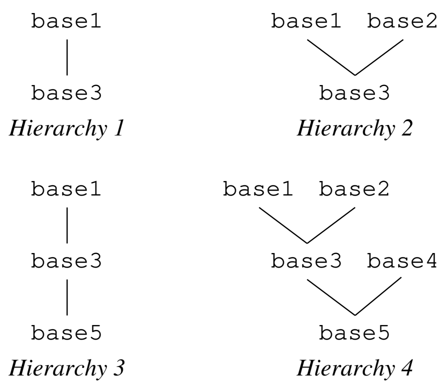

Figure 4. KB hierarchies

The different variants of the `UserInput` process define particular combinations of user actions on individual KBs. By the nature of the CO_4 protocol, this is enough to constrain the possible scenarios throughout the whole KB hierarchy. The user environments that we have considered typically consists of the subscription phase (all members register to their respective group) followed by one or two transactions (e.g. submission and votes of one proposal). We have also defined more "realistic"users (any of a fairly representative sample of possible user actions, any number of times), but for interactive simulation only since they produce an infinite state space.

For systems of this complexity, state space explosion quickly becomes intractable despite all measures taken to limit it. Furthermore, due to the number and complexity of data structures involved, the memory consumption for each state is important, further limiting the achievable graph sizes: on a *Sun UltraSparc* station with 256Mb RAM, the ceiling is about 350,000 states. Table 2 gives the graph generation time and size, before and after minimization, for some scenarios (e.g. "S+1P" means "all users subscribe, then one proposal is submitted to vote"). The number of transitions ranges between 1.3 and 2.4 times the number of states, increasing with the complexity of the hierarchy.

specification		CPU	#states	
hierarchy	user input	time (s)	before	after
1	S+1P	12.7	133	47
1	S+2P	30.8	8 185	1 552
2	S+1P	45.9	3 568	593
3	S+1P	8:02.2	35 002	3 619
4	1P	1:19:20.3	62 557	3 316

Table 2. Graph generation statistics

In this article, we can only provide an intuitive and informal survey of our verification results. More detailed data, including diagnostic traces, can be found in [22].

4.3 Properties of Co4

There is no such thing as a "valid system" in the absolute: one always checks validity w.r.t. some expected properties, either general ones (e.g. absence of deadlock) or case-specific ones (e.g. consistency of KBs). In the case of CO_4, [5] lists five such properties:

0 (Intelligibility) For each message sent there is a rule triggered by that kind of message.

1 (Base Consistency) Assuming that concurrent proposals are independent, the knowledge stored in group bases never reaches an inconsistent state.

2 (Liveness and Fairness) The subscribers can submit proposals to their group base at any moment.

3 (Consensus) Any submission is accepted if and only if all the subscribers accept it.

4 (Termination) Under several finiteness assumptions, any submission is eventually rejected or accepted.

All these properties have been addressed to some extent in the work presented here. Properties 0, 1 and 3 have been the object of a specific attention; this is detailed in the following sections. Strictly speaking, property 2 is not valid: an individual base will not accept a new submission while it is processing some other rule. In a looser sense, all the tested scenarios show that a registered individual base always eventually accepts submissions from its user. Property 4 is also partially covered by deadlock detection.

4.4 Deadlock Detection

The most straightforward property (but not the simplest, in terms of computation cost) that can be checked on a specification is the existence of deadlocks, i.e. states from which no transition is possible. The interpretation of such states depends on the specification: deadlocks are not necessarily errors.

In the case of CO_4, we have conceived the LOTOS specification in such a way that deadlocks indeed correspond to problems. In particular, normal end of the considered scenario does not lead to deadlock but rather to a state where the system continuously "rings" an OK action. However, this occurs only if all KBs are idle (no pending transaction) and all queues are empty: other situations indeed produce a deadlock.

According to this, property 0 amounts to absence of deadlocks: if there is no rule for some message then the receiving base will deadlock. Property 4 is also related to deadlocks: if some submission remains pending then the concerned KBs will not terminate—either they will stop (deadlock) or they will keep exchanging messages forever (livelock). We have not dealt with the latter case.

Technically, the existence of deadlocks is reported by CÆSAR at the issue of exhaustive model generation. EXHIBITOR can then be applied to the generated model in order to report traces leading to those states. If exhaustive generation is not possible, deadlocks can still be found by applying EXHIBITOR to the specification.

Deadlock detection has lead to the detection of two kinds of problems: *lack of asynchronism* and *unexpected receptions*. Both are complementary and to some extent adequately handled by the CO_4 protocol, but at least deserve some further comments.

4.4.1 Lack of Asynchronism

In practice, CO_4 messages are carried by an asynchronous reliable transport service (though this is not stated explicitly in [5]). On the other hand, asynchronism in the specification increases the state space explosion. In a first approach, we have therefore produced a strongly synchronized specification, in which messages are exchanged by direct rendezvous between KBs.

Several deadlocks were detected on this specification, all related to "dialogue of the deaf" situations, that is, two bases simultaneously trying to send a message to each other while not willing to receive. These deadlocks do not result from the CO_4 protocol but rather from an inadequate modelling of interactions: they disappear when buffering is introduced in the specification.

On the other hand, to limit state space blow-up we have introduced as little buffering as possible: queues have been inserted for upward traffic (from members to group) while downward traffic remains synchronous. This hybrid solution is a compromise between state space explosion and excessive abstraction, and has been found to be sufficient to avoid the previous deadlocks. Table 3 illustrates the increase in model size caused by the introduction of these buffers. These results were obtained on hierarchy 1, n being the number of submitted proposals.

specification	n	#states	#trans
No queues	1	38	49
	2	339	442
	3	3202	4158
With queues	1	133	196
	2	8 185	13 338
	3	>300 000	>500 000

Table 3. Model sizes with and without queues

4.4.2 Unexpected receptions

The second kind of deadlocks that were found on the specification had to do with unexpected message receptions, i.e. messages that were received without any rule for handling them, and therefore refuting property 0. A couple of cases are really missing rules (or variants of rules) in the source description, that were reported as such to the designers of

CO$_4$. Most cases, however, result from buffering in communications: messages pertaining to some transaction can be received after that transaction has been completed (or cancelled). For example, a group base may receive a vote for a proposal that has been withdrawn by its initiator in the meantime.

As a matter of fact, [5] states that "the non recognized messages are just ignored". Taking such a statement as a default rule, then property 0 is trivially satisfied and our specification should not deadlock in such cases. However, we feel that applying blindly this kind of rule is a dangerous thing to do as part of a design process. It is very useful to know precisely which unexpected messages are received, in order to decide whether they can be safely ignored. In particular, the really missing rules would have been much more difficult to find, had the corresponding messages been silently discarded by our model.

The final version of our specification takes an intermediate position: unexpected messages are indeed discarded, but produce a monitoring event on gate `signal`. A search for traces leading to such events using EXHIBITOR has demonstrated four different cases of unexpected message reception.

4.5 Consistency and Consensus

Consistency of knowledge bases is a condition on the internal state of the system. To be able to observe this state, the specification produces `signal` events whenever knowledge is modified, with one attribute indicating the consistency of the modification. EXHIBITOR can then be used to search for traces leading to that event.

Consistency (property 1) requires two concurrent submissions p and p' to be independent, meaning that the consistency of p w.r.t. a repository K is preserved if p' is added to K. This is an important limitation of the protocol, since it requires "rational" users that can detect and avoid contradictory proposals. Indeed, we have found sequences leading to inconsistent bases in scenarios where this hypothesis is not ensured. This problem is acknowledged as a main target for improvement by the designers of CO$_4$. In this respect, the LOTOS specification will be useful as a prototype to experiment alternative solutions.

Another similar but unexpected consistency problem has been detected: if two users ask (through their individual base) the same group base to subscribe to different higher-level groups, there are cases where both transactions can be successfully fulfilled, leading to anomalous situations (and specification deadlocks) afterwards. To search such situations, `signal` events have been associated to successful registration in a group base, and we considered a variant of hierarchy 3 with a new `base6` besides `base5`.

The consensus property has been verified too, by using EXHIBITOR to search for traces where a submitted proposal was accepted (a `signal` event is produced when the repository is modified) without accepting vote from some member of the group. No such traces were found on any of the tested scenarios.

5 Conclusion

After a long history of continuous improvements and more and more ambitious applications, formal verification techniques based on model checking now reach maturity: thanks to the computing power of modern computers and the level of sophistication of verification tools, significant results can be obtained for real-size systems such as CO$_4$.

The formal specification of the CO$_4$ protocol ranges among medium-size LOTOS specifications (2400 lines of standard LOTOS code). To obtain finite state spaces of tractable size, the specification considers a fixed number of entities (2 to 5 bases) and particular execution scenarios only.

The CADP verification toolbox has been used to verify some expected properties of the protocol, such as the consistency of the knowledge bases or the absence of unexpected message reception. More precisely, we searched for traces leading to violation of these properties using the EXHIBITOR tool. Several were found: some of them were known, others were revealed by the verification. Even in the former case, model checking helps to analyze the conditions in which they occur and provides a basis to study possible remedies.

This study has been conducted in close collaboration with the designers of CO$_4$. All the verification results were reported to them, as well as numerous questions, corrections and remarks concerning the definition of the protocol in [5]. These comments have been taken into several successive revisions of the document [6].

The formal verification work on CO$_4$ does not end with the publication of these results. The LOTOS specification will be used to study new improvements of the protocol, either by interactive exploration with XSIMULATOR or by more exhaustive analysis with other CADP tools. Another promising approach lies in a closer integration between verification and implementation: a novel feature of the CÆSAR compiler, still in development, compiles the specification into an executable prototype, interfaced with user-provided code [10].

6 Acknowledgements

This study has largely benefitted from a close collaboration with Jérôme Euzenat and Loïc Tricand de la Goutte, designers of the CO$_4$ protocol, and with Hubert Garavel and the CADP team for the hottest improvements to their tools. Many thanks too to Radu Mateescu, Mihaela Sighireanu and the anonymous reviewers for their judicious comments on this text. Some computations have been gracefully hosted on a multi-processor Sparc server of the ENSIMAG school of Grenoble. Norman Ramsey's NOWEB literate programming system has been used to write the specification.

References

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

[2] 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.

[3] 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.

[4] Jérôme Euzenat. Building Consensual Knowledge Bases: Context and Architecture. In *Proceedings of the 2nd International Conference on Building and Sharing Very Large-Scale Knowledge Bases (KBKS), Enschede the Netherlands*, pages 143–155, 1995.

[5] Jérôme Euzenat. Building consensual Knowledge Bases: Protocol. Unpublished, January 13 1997.

[6] Jérôme Euzenat. A Protocol for Building Consensual and Consistent Repositories. Research report, INRIA, 1997. To appear.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

[16] ISO/IEC. LOTOS Description of the CCR Protocol. Technical Report 11590, International Organization for Standardization — Open Systems Interconnection, Genève, 1995.

[17] 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.

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

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[20] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[21] 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.

[22] Charles Pecheur. Specification and Validation of the CO4 distributed knowledge system using LOTOS. Research report, INRIA, 1997. To appear.

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