INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Specification and Verification of the Co4 Distributed Knowledge System Using LOTOS

Charles Pecheur

N° 3259

September 1997

THÈME 1

Rapport de recherche

ISSN 0249-6399

Specification and Verification of the Co4 Distributed Knowledge System Using LOTOS

Charles Pecheur*

Thème 1 — Réseaux et systèmes
Projet VASY

Rapport de recherche n3259 — September 1997 — 97 pages

Abstract: This report presents the 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. The full commented Lotos specification and excerpts from detailed results are included in appendices.

Key-words: Formal Methods, Verification, Model Checking, Lotos, Knowledge Systems.

(Résumé : tsvp)

Short version of this report in "Specification and Verification of the Co4 Distributed Knowledge System using LOTOS", in: Proceedings of the 12th IEEE International Conference on Automated Software Engineering, Incline Village, Nevada, USA, November 1997.

* `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

Spécification et vérification du système de connaissances distribué Co4 en LOTOS

Résumé : Ce rapport relate la spécification et la vérification d'un protocole de décision consensuelle utilisé dans Co4, un environnement informatique dédié à la construction d'une base de connaissances distribuée. Ce protocole a été spécifié en LOTOS, une technique de description formelle normalisée par l'ISO. Les outils CADP de la boîte à outils EUCALYPTUS ont été utilisés pour vérifier différentes propriétés de sûreté et de vivacité. Ce travail de vérification a confirmé une violation connue de consistance de la connaissance et a mis en évidence un cas de hiérarchie inconsistante, quatre cas de réception non prévue de message et plusieurs autres corrections locales dans la définition du protocole. La spécification LOTOS complète et commentée et certains extraits des résultats détaillés figurent en appendices.

Mots-clé : Méthodes Formelles, Vérification, LOTOS, Systèmes de Connaissances.

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 only applicable to systems with a finite state space.

We relate here a case of application of formal verification in the field of distributed knowledge bases. Co_4 is a computer environment dedicated to the incremental and concurrent building of a knowledge base [Euz95]. In particular, communication between Co_4 entities follow a consensual decision protocol derived from peer-reviewing policies.

This protocol has been specified in the Iso formal description technique Lotos [ISO88], and the Cadp (Cæsar/Aldébaran) toolset [Gar96] has been used to verify different expected properties of 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 report 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. Section 4 presents the verification tools, methodology and results. Appendix A contains the complete text of the Lotos specification. Appendix B provides some representative samples of the detailed verification results.

2 The LOTOS Language

Lotos [ISO88] 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 report does not assume familiarity with Lotos from the reader. The short following overview be sufficient to understand the essence of the forthcoming excerpts. Tutorials for Lotos are available, e.g. [BB88, Tur93].

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. As a matter of fact, most data types of our specification are defined in a much more concise and readable extended syntax [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. Behaviours can manipulate data values and exchange them through their interactions. The main operators of LOTOS are summarized in Table 1.

`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

LOTOS has been applied to many complex systems such as OSI services and protocols [ISO89, ISO95], but also cryptographic protocols [LBK+96] and hardware components [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. The tools used in the present work 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 sub-section is divided into a *presentation* part, which describes some aspect of CO_4, and a *specification* part, which explains how it is expressed in 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 [Euz97a].

The LOTOS specification is intended for model-based verification using the CADP verification 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 has been taken into account when writing the specification: we define 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 practice most problems can be found on small systems alone.

Most data types have been defined using the syntax extensions provided by the APERO pre-processor [Pec96]. These extensions provide convenient concise declarations for many common data structures such as records, enumerations, sets, lists, or even ML-style "algebraic" types [MTH90]. 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 needed to allow the compilation of algebraic data type definitions.

The complete specification (with APERO notations) is about 1400 lines long and can be found in Appendix A; only small significant excerpts are given here, in which some technical details have been wiped out for the sake of clarity. In particular, ellipses (...) are used to indicate omitted parts; they do not belong to the original LOTOS specification.

3.1 Top-Level Structure

Presentation A Co_4 system is made up of a collection of *(knowledge) bases*, or KBs, organized as a tree, as illustrated on Figure 1. The leaf nodes are *individual* KBs directly connected to the external world, and holding the knowledge of a single human user (or a group thereof). Intermediate nodes are *group* KBs; 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[1].

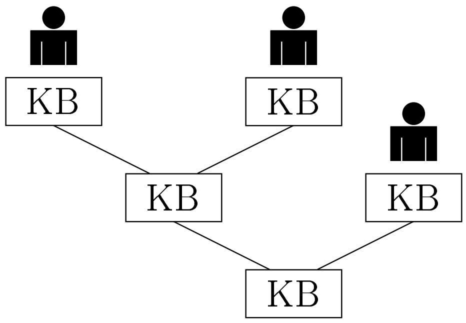

Figure 1: A Co4 hierarchy

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.

[1] An individual KB may even turn into a group KB or vice-versa, but for simplification purposes the LOTOS specification considers static hierarchies only.

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 bases from communications with parent bases[2]. 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.

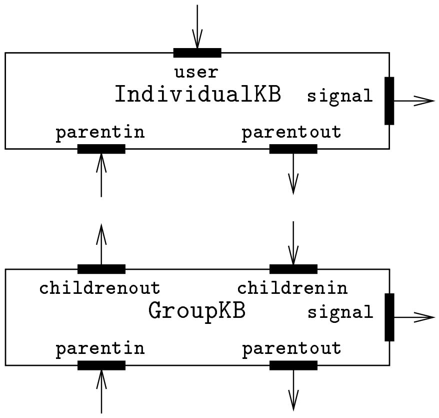

Figure 2: Interface of knowledge bases

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

[2]This separation eliminates unwanted synchronizations (e.g. message to parent received by a child), as part of our efforts to limit the cost of the verification work.

```
  |[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:

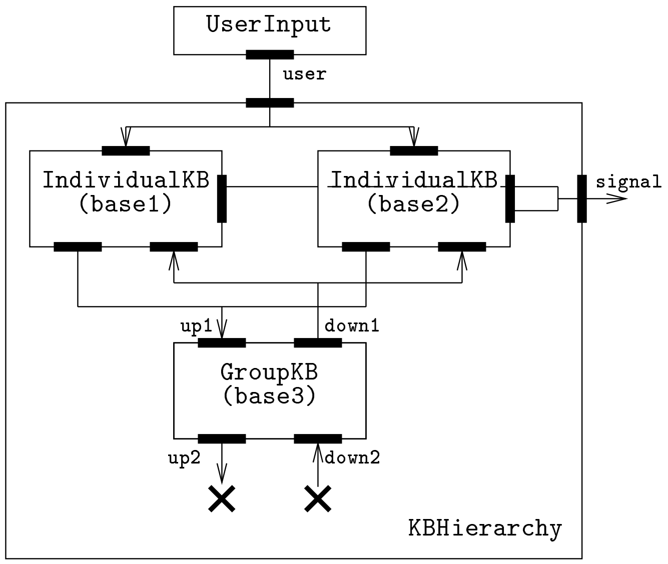

Figure 3: Top-level specification with three bases

```
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 data structures, which we can sort out in four categories:

- *Identifiers* (e.g. *message identifiers*[3] 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. [Euz97a] 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 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, or

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

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 keeps the possibility to study the behaviour of the system in the case of contradictory concurrent proposals. This also allows to study how the protocol can be modified to preserve consistency in this case. We specify consistency as a binary *compatibility relation* over atoms, such that a base is consistent if it does not contain incompatible atoms.

[3] called *surrogates* in [Euz97a].

3.3 The Co4 Protocol

Presentation The principles underlying the Co4 protocol are those of peer-reviewing: before being committed, 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 registration of new members and other transactions. In addition, any request can be cancelled by its author before it is accepted.

[Euz97a] describes Kbs as input-output automata with state variables, and defines the protocol in terms of transitions of these automata, with rules of the following general form:

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

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; details vary from rule to rule. Each type of base (individual and group) has its own set of rules, for a total of 35. As a concrete example, the rule *handle-notify* is defined as

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

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 (where "_" denotes "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 is 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 [Euz97a] was 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. Representative samples of the detailed diagnostics provided by the tools are given in Appendix sec:detailedresults.

4.1 LOTOS Tools

All the processing of LOTOS specifications 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. Besides the APERO data type pre-processor described in Section 3, the EUCALYPTUS toolset also contains CADP[4], 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 used the following CADP tools:

- CÆSAR [GS90] and CÆSAR.ADT [Gar89] are compilers that transform respectively the control and data part of a LOTOS program into a state graph[5] describing its exhaustive behaviour. 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 [FKM93] is a verification tool for comparing or minimizing graphs with respect to any of several simulation and bisimulation relations [Par81, Mil89]. Initially designed to deal with explicit graphs, it has been extended to 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 to walk through the different alternative branches of the graph, using back and forth step-by-step execution.
- 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. It also 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.

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 was used for a first interactive exploration of the LOTOS specification. This simulation allowed to detect some missing rule in Co_4. When the most obvious problems have been fixed, we turned to more systematic exploration techniques.

For the majority of scenarios, the complete graph was generated with CÆSAR and minimized modulo observational equivalence [Mil89] with ALDÉBARAN. We then applied EXHIBITOR to the obtained graph to search for traces leading to particular situations: deadlock

[4]CÆSAR/ALDÉBARAN Development Package

[5]This kind of graph is called a *Labelled Transition System* (LTS for short), and is the *model* of the specification. We will use the words “model” and “graph” in this report.

states, knowledge inconsistencies, etc. In more complex scenarios, where the graph is too big to be completely generated, EXHIBITOR was applied to the LOTOS specification itself, using on-the-fly graph generation. XSIMULATOR was still used to re-play the sequences found and get a better understanding of what is going on.

We 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 on Figure 4.

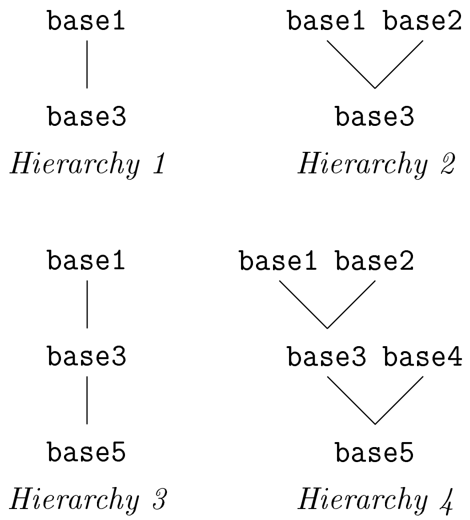

Figure 4: KB hierarchies

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 considered typically consist 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 256 Mb RAM, the ceiling is about 350,000 states. Table 2 gives the graph generation time and size, before and after minimization, for some scenarios. 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	subscription + one proposal	12.7	133	47
1	subscription + two proposals	30.8	8 185	1 552
2	subscription + one proposal	45.9	3 568	593
3	subscription + one proposal	8:02.2	35 002	3 619
4	one proposal	1:19:20.3	62 557	3 316

Table 2: Graph generation statistics

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 CO4, [Euz97a] 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 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 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, deadlocks can be detected by ALDÉBARAN by simple inspection of the generated graph. EXHIBITOR can then be applied to the generated model in order to report traces leading to those states. If exhaustive generation is not possible, EXHIBITOR can still find deadlocks using on-the-fly exploration of the LOTOS 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 was not stated explicitly in [Euz97a]). 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 rendez-vous 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. An example is given in Appendix B.1. 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 few 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.

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 were really missing rules (or variants of

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

rules) in the source description, which were reported as such to the designers of Co_4. Most cases, however, were related to the asynchronous communications: messages pertaining to some transaction can be received after that transaction has been completed (or cancelled). Appendix B.2 shows an example of this.

As a matter of fact, [Euz97a] 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 believe 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. One such sequence is shown in Appendix B.3. 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 bases) 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 sequence found is 40 transitions long and is shown in Appendix B.4.

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 [Euz97a]. These comments have been taken into several successive revisions of the document [Euz97b].

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 [GJM+97].

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 Hubert Garavel, Radu Mateescu and Mihaela Sighireanu 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 LOTOS specification.

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.

[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.

[Euz95] 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.

[Euz97a] Jérôme Euzenat. Building consensual Knowledge Bases: Protocol. Unpublished, January 13 1997.

[Euz97b] Jérôme Euzenat. A Protocol for Building Consensual and Consistent Repositories. Research Report RR-3260, INRIA, September 1997.

[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.

[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.

[ISO95] ISO/IEC. LOTOS Description of the CCR Protocol. Technical Report 11590, International Organization for Standardization — Open Systems Interconnection, Genève, 1995.

[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.

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

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Par81] 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.

[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.

[Pec97a] Charles Pecheur. Erratum for the CO4 protocol. Internal note, February 18 1997.

[Pec97b] Charles Pecheur. Erratum for the CO4 protocol - part 2. Internal note, March 4 1997.

[Pec97c] Charles Pecheur. Erratum for the CO4 protocol - part 3. Internal note, April 29 1997.

[Pec97d] Charles Pecheur. Remarks and Questions about the CO4 protocol. Internal note, February 3 1997.

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

A Specification of the Co4 protocol in LOTOS version 3

A.1 Introduction

CO$_4$ is a computer environment dedicated to the incremental and concurrent building of a knowledge base that organizes the manipulation and distribution of media related to that knowledge among the participating people. In particular, CO$_4$ features a consensual decision protocol derived from peer-reviewing policies. This document contains the specification of the CO$_4$ protocol in the ISO specification language LOTOS [ISO88].

Notational convention The full LOTOS code is provided, in the form of labelled chunks like the following sample[6]:

21a ⟨*sample* 21a⟩≡

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

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

21b ⟨*other sample* 21b⟩≡

```
(* ... *)
⟨sample 21a⟩
(* ... *)
```

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

The general principles of CO$_4$ are described in [Euz95]. The CO$_4$ protocol is more precisely defined in [Euz97a], which is the reference input document used for producing this LOTOS specification.

The writing of this specification has necessitated a thorough and careful analysis of [Euz97a]. As a result of this reading, several imprecisions or inconstistencies have been put forth and corrected in collaboration with the designers of CO$_4$. The remarks and corrections w.r.t. [Euz97a] are listed in documents [Pec97a, Pec97d, Pec97b, Pec97c]. As a result, an improved version of [Euz97a] has been produced [Euz97b].

Validation tools This specification is intended for analysis using the CADP validation tools [Gar96]. This has some consequences in the way it is written:

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

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

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

A.2 History

Before this public version, the LOTOS specification has undergone several important reorganizations. This section briefly describes the major versions.

A.2.1 Version 1

This version covers only single-level hierarchies — propagation of proposals and votes between multiple levels is not specified.

Communication between bases uses direct rendez-vous. This leads to many deadlock situations due to two bases simultaneously trying to send messages to each other.

A.2.2 Changes in Version 2

To avoid the deadlocks of version 1, version 2 introduces queues between bases, and merges all message types into a single sort `Message`.

A.2.3 changes in version 3

Version 3 completes the coverage of the CO_4 protocol — only challenging of forwarded proposals is still under design and therefore not covered in this specification.

A.3 Data Structures

This section discusses the data type part of the specification.

A.3.1 Identifiers

This section describes the various kinds of identifiers used in CO_4.

Base identifiers Identifiers for bases. Represented as s, r, g, G, B in [Euz97a]. Described as a flat name space, with a few defined constants.

23a ⟨*data types* 23a⟩≡

```
nametype        BaseId is
name            BaseId
endtype

type            BaseIdOpns is BaseId
opns            base0,base1,base2,base3,base4,
                base5,base6,base7,base8,base9 : -> BaseId
eqns
ofsort BaseId
        base0 = first ;
        base1 = next(base0) ;
        base2 = next(base1) ;
        base3 = next(base2) ;
        base4 = next(base3) ;
        base5 = next(base4) ;
        base6 = next(base5) ;
        base7 = next(base6) ;
        base8 = next(base7) ;
        base9 = next(base8) ;
endtype
```

Defines:
`BaseId`, used in chunks 27, 28b, 30b, 33–35, 37–39, 42–45, 47b, 51, 52, 56, 58b, 63, 65a, and 69–74.

Surrogates Surrogates are identifiers attributed locally by bases to pending proposals and votes, represented as n or m in [Euz97a]. There are two disjoint classes of surrogates:

- *Request identifiers* are generated by the initiator of a request; they point to entries in table A of the initiator.
- *Call for comments (CFC) identifiers* are generated by the group base when a vote is started; they point to entries in table C of the base.

It would be sensible to specify these classes as separate Lotos sorts – indeed, it was done that way until version 2. However, both classes may appear in a common context (namely, the `senderid` field of an entry in state table C). For this reason we choose to specify them as a single sort `Id`. This is a flat, infinite data type as provided by the Apero `nametype` shorthand. The Cadp data type compiler recognises such data types as isomorphic to natural numbers and represents them very efficiently as built-in integers.

23b ⟨*data types* 23a⟩+≡

```
nametype        Id is
name            Id
endtype
```

Defines:
`Id`, used in chunks 28, 30b, 31, 33b, 37, 38, 46, 47, 49, 50, 52–56, 62–65, and 67.

A.3.2 Knowledge Data

Co$_4$ manipulates knowledge data: each node in the system manages a *repository*, that can be modified according to *proposals*. The precise structure of knowledge data is transparent to the Co$_4$ protocol. Co$_4$ knows of three operations over knowledge:

$K + p$ applies proposal p to repository K,

$p * p'$ combines proposals p and p' into a single proposal,

$K?p$ tests whether proposal p is applicable to repository K.

The latter is related to the notion of consistency: a proposal p is applicable to repository K iff $K + p$ is consistent.

[Euz97a] does not define precisely the semantics of the merge operator $p * p'$. We take the following interpretation: suppose that an initial proposal p is challenged by a proposal p'. We assume that p' describes proposals w.r.t. the effect of p, whereas $p * p'$ describes the cumulated proposals w.r.t. the original repository.

[Euz97a] assumes that “concurrent proposals are independent” (assumption (d) p. 22), meaning that if each of them is applicable to a repository, then they can be applied together in any order. Formally, for two concurrent proposals p and p' and for any repository K, the following property must hold:

$$K?p \wedge K?p' \Rightarrow (K + p)?p' \wedge (K + p')?p$$

This is a rather strong and unrealistic hypothesis; it requires “rational” users that can detect and avoid contradictory proposals.

We do not enforce this possibility at the level of the Lotos specification. This keeps the possibility to study the behaviour of the system in the case of contradictory concurrent proposals, in order to get a clear understanding of the potential consistency violations and to test how the protocol can be modified to avoid those violations.

We adopt the following simple model:

- A *repository* K is a set of *atoms* $\{a_1, \ldots, a_n\}$.
- There is a binary *compatibility relation* $a?a'$ over atoms. A base is consistent if it does not contain incompatible atoms.

- A *change* c is either an *assertion* $assert(a)$ or a *retraction* $retract(a)$. Applying an assertion (resp. retraction) to a repository adds (resp. deletes) the corresponding atom.

- A *proposal* p is a sequence $(c_1, \ldots, c_n)$ of assertions. Merging proposals amounts to concatenating the sequences, and applying a proposal to a repository amounts to applying the changes in the given order.

We define three atoms `white`, `black` and `square`, where `white` and `black` are mutually incompatible.

25a ⟨*data types* 23a⟩+≡

```
enumtype        Atom is
enum            white, black, square : Atom
endtype

type            AtomCompatibility is Atom, Boolean
opns            comp : Atom, Atom -> Bool
eqns forall a1, a2 : Atom
ofsort Bool
        comp(black, white) = false ;
        comp(white, black) = false ;
        comp(a1, a2) = true ;
endtype
```

Defines:
`Atom`, used in chunk 25b.

The rest is a direct translation of our informal description above.

25b ⟨*data types* 23a⟩+≡

```
settype         Knowledge is Atom
set             Knowledge
elements        Atom
endtype

datatype        Change is Atom
datasorts       <eq> Change =
                <is> assert(Atom) |
                <is> retract(Atom)
endtype

stringtype      Proposal is Change
string          Proposal
elements        Change
endtype
```

```
type            KnowledgeOpns is
                AtomCompatibility, Knowledge, Proposal, Boolean
opns            atom            : Change -> Atom
                _+_             : Knowledge, Change -> Knowledge
                _+_             : Knowledge, Proposal -> Knowledge
                _*_             : Proposal, Proposal -> Proposal
                query           : Knowledge, Atom -> Bool
                query           : Knowledge, Proposal -> Bool
                isconsistent    : Knowledge -> Bool
                assert          : Knowledge -> Proposal
eqns forall
        k : Knowledge,
        a, a1, a2 : Atom,
        c : Change,
        p, p1, p2 : Proposal
ofsort Atom
        atom(assert(a)) = a ;
        atom(retract(a)) = a ;
ofsort Knowledge
        k + assert(a) = insert(a, k) ;
        k + retract(a) = remove(a, k) ;
        k + <> = k ;
        k + (c + p) = (k + c) + p ;
ofsort Proposal
        p1 * p2 = p1 ++ p2 ;
ofsort Bool
(* NB: uses hidden constructor insert_C generated for CADP *)
        query({}, a) = true ;
        query(insert_C(a1, k), a2) = comp(a1, a2) and query(k, a2) ;
        isconsistent({}) = true ;
        isconsistent(insert_C(a, k)) = query(k, a) and isconsistent(k) ;
        query(k, p) = isconsistent(k + p) ;
ofsort Proposal
        assert({}) = <> ;
        assert(insert_C(a, k)) = assert(a) + assert(remove(a, k)) ;
endtype
```

Defines:
`Change`, never used.
`Knowledge`, used in chunk 38.
`Proposal`, used in chunks 27, 28, 31, 45–47, 53, 58a, 60, 62, 68, 71c, 72, and 74a.
Uses `Atom` 25a.

A.3.3 Messages

This section defines data structures for the messages exchanged in the Co$_4$ protocol. These formats are formally defined in [Euz97a]. The different kinds of messages are characterized by a so-called *performative*, which defines the purpose of the message and the structure of its content.

We define messages as a single LOTOS sort `Message`, with one constructor for each performative. Before defining messages themselves, we need a few auxiliary types.

First, we define a *request* as a "subject of conversation". They correspond directly to message types. Requests will be stored in KB tables and put in forward messages. Note that the *forward* variant is recursive, i.e. it contains an encapsulated request. Since this type does not have a conventional structure (record, union, ...), it is defined using the more flexible APERO `datatype` facility.

27 ⟨*data types* 23a⟩+≡

```
datatype        Request is BaseId, Proposal
datasorts       <eq> Request =
                <is> reqregister(BaseId) |
                <is> reqevaluate(Proposal) |
                <is> reqachieve(Proposal) |
                <is> reqforward(Request) |
                <is> reqchallenge(Proposal) |
                <is> reqtell(Proposal)
endtype

type            RequestOpns is Request
opns            content         : Request -> BaseId
                content         : Request -> Proposal
                content         : Request -> Request
eqns forall     b : BaseId,
                p : Proposal,
                r : Request
ofsort BaseId
        content(reqregister(b)) = b ;
ofsort Proposal
        content(reqevaluate(p)) = p ;
        content(reqachieve(p)) = p ;
        content(reqchallenge(p)) = p ;
        content(reqtell(p)) = p ;
ofsort Request
        content(reqforward(r)) = r ;
endtype
```

Defines:
`Request`, used in chunks 28b, 30b, 31, 33b, 46b, 47a, 49a, 52–54, 56, 58b, 62–64, 68, 70–72, and 74a.
Uses `BaseId` 23a and `Proposal` 25b.

Second, *answers* are used as the contents of *reply* and *notify* messages. An answer is either *accept*, *reject*(r) or *challenge*(p), where r motivates the rejection but is transparent to the protocol and will be ignored here, and p is a proposal.

28a ⟨*data types* 23a⟩+≡

```
datatype        Answer is Id, Proposal
datasorts       <eq> Answer =
                <is> acceptx |  (* NB: accept is a LOTOS keyword *)
                <is> reject |
                <is> challenge(replywith : Id, content : Proposal)
endtype

type            AnswerOpns is Answer, Boolean
opns            answer : Bool -> Answer
eqns
ofsort Answer
        answer(true) = acceptx ;
        answer(false) = reject ;
endtype
```

Defines:
`Answer`, used in chunks 28b, 47b, 56, and 63.
Uses `Id` 23b and `Proposal` 25b.

Now comes the definition of all messages, as a single APERO `datatype` declaration. The source and destination address are not defined as proper parts of the message; they will be passed as separate attributes of LOTOS events. Some embedded surrogates and base identifiers (e.g. in *askall* or *forward* messages) are not shown in the specification. They are irrelevant to the Co_4 protocol, though they provide information for the user of individual bases.

28b ⟨*data types* 23a⟩+≡

```
datatype        Message is
        Id, BaseId, Proposal, Request, Answer
datasorts       <eq> Message =
        (* upward messages *)
                <is> register(Id, BaseId) |
                <is> evaluate(Id, Proposal) |
                <is> achieve(Id, Proposal) |
                <is> forward(Id, Request) |
                <is> deny(Id) |
                <is> reply(Id, Answer) |
        (* downward messages *)
                <is> askall(Id, Request) |
                <is> error(Id) |
                <is> notify(Id, Answer) |
```

```
                <is> poolnotify(Id, Answer) |
                <is> pooldeny(Id) |
                <is> tell(Proposal)
endtype

type            MessageOpns is Message
opns            replywith :     Message -> Id
                inreplyto :     Message -> Id
                content :       Message -> BaseId
                content :       Message -> Proposal
                content :       Message -> Request
                content :       Message -> Answer
eqns forall
        m : Id,
        n : Id,
        b : BaseId,
        p : Proposal,
        r : Request,
        a : Answer
ofsort Id
        replywith(register(m, b)) = m ;
        replywith(evaluate(m, p)) = m ;
        replywith(achieve(m, p)) = m ;
        replywith(forward(m, r)) = m ;
        replywith(askall(n, r)) = n ;
        inreplyto(deny(m)) = m ;
        inreplyto(error(m)) = m ;
        inreplyto(notify(m, a)) = m ;
        inreplyto(reply(n, a)) = n ;
        inreplyto(pooldeny(n)) = n ;
        inreplyto(poolnotify(n, a)) = n ;
ofsort BaseId
        content(register(m, b)) = b ;
ofsort Proposal
        content(evaluate(m, p)) = p ;
        content(achieve(m, p)) = p ;
        content(tell(p)) = p ;
ofsort Request
        content(forward(m, r)) = r ;
        content(askall(n, r)) = r ;
ofsort Answer
        content(notify(m, a)) = a ;
        content(reply(n, a)) = a ;
        content(poolnotify(n, a)) = a ;
endtype
```

Defines:
`Message`, used in chunks 35b, 42–44, 51, 69a, and 74a.
Uses `Answer` 28a, `BaseId` 23a, `Id` 23b, `Proposal` 25b, and `Request` 27.

A.3.4 External Interactions

This section defines data types used in interactions with the external world. Those interactions use mixed in/out attribute patterns. The data type defined here is an enumeration of tags characterizing the different kinds of user actions.

- `UserAction` values are associated with events on gate `user`, which represent human-driven initiatives in individual bases.
- `SignalVal` values are associated with events on gate `signal`, which are introduced to be able to monitor certain internal conditions in the system.

30a ⟨*data types* 23a⟩+≡

```
enumtype        UserAction is
enum            doregister, doevaluate, doachieve, doforward, dodeny,
                doaccept, doreject, dochallenge : UserAction
endtype

enumtype        SignalVal is
enum            sigregistered, sigstored,
                signotinA, signotinP, signotinC, signotconsistent
                : SignalVal
endtype
```

Defines:
`SignalVal`, never used.
`UserAction`, used in chunk 74a.

A.3.5 State Variables

This last part of the data types defines the data structures used for state variables, as described p.8 in [Euz97a].

Most of these variables are sets of tuples, which are indeed used as tables indexed by surrogates. We specify them as such, using Apero `tabletype` definitions.

Submitted requests This is variable A in [Euz97a]. In the case of group bases, entries contain more data identifying the origin of the submission (i.e. who sent the forward). we therefore define two different data types for A: `SubmittedTbl` and `GrpSubmittedTbl`.

30b ⟨*data types* 23a⟩+≡

```
recordtype      SubmittedEntry is Id, Request
record          submitted : SubmittedEntry
fields          id : Id
                req : Request
endtype

tabletype       SubmittedTbl is SubmittedEntry
table           SubmittedTbl
elements        SubmittedEntry
key             id : Id
endtype

recordtype      GrpSubmittedEntry is Id, Request, BaseId
record          submitted : GrpSubmittedEntry
fields          id : Id
                req : Request
                sender : BaseId
                senderid : Id
endtype

tabletype       GrpSubmittedTbl is GrpSubmittedEntry
table           GrpSubmittedTbl
elements        GrpSubmittedEntry
key             id : Id
endtype
```

Defines:
`GrpSubmittedTbl`, used in chunk 38.
`SubmittedTbl`, used in chunks 37 and 45a.
Uses `BaseId` 23a, `Id` 23b, and `Request` 27.

Pending votes This is variable P in [Euz97a]. An auxiliary enumerated type `Status` defines the four possible statuses of a pending vote: @, A, R and C in [Euz97a].

31 ⟨*data types* 23a⟩+≡

```
enumtype        Status is
enum            new, accepted, rejected, challenged : Status
endtype

recordtype      PendingEntry is Id, Request, Status
record          pending : PendingEntry
fields          id : Id
                req : Request
```

```
                status : Status
endtype

tabletype       PendingTbl is PendingEntry
table           PendingTbl
elements        PendingEntry
key             id : Id
endtype

type            PendingTblOpns is PendingTbl, RequestOpns
opns            searchproposal  : Proposal, PendingTbl -> Bool
                getproposal     : Proposal, PendingTbl -> PendingEntry
eqns forall     e : PendingEntry,
                t : PendingTbl,
                p : Proposal
ofsort Bool
        searchproposal(p, {}) = false ;
        is_reqachieve(req(e)),
        content(req(e)) eq p =>
                searchproposal(p, insert_C(e, t)) = true ;
        searchproposal(p, insert_C(e, t)) =
                searchproposal(p, t) ;
ofsort PendingEntry
        is_reqachieve(req(e)),
        content(req(e)) eq p =>
                getproposal(p, insert_C(e, t)) = e ;
        getproposal(p, insert_C(e, t)) =
                getproposal(p, t) ;
endtype
```

Defines:
`PendingTbl`, used in chunks 37 and 38.
Uses `Id` 23b, `Proposal` 25b, and `Request` 27.

Accepted and ignored modifications Variables M and L contain approved proposals that are or are not accepted by a base, but do not influence the behaviour of the protocol. We therefore do not specify them. At the behaviour level, it is still possible to produce an external action indicating storage in M or L.

Knowledge repository Already defined in a previous section.

Subscribers This is variable S of [Euz97a], defined as an APERO `settype` definition. We add an operation `pick` that returns the "first" element of a set[7].

[7]Note that this definition is acceptable under the evaluation strategy of CADP, but corrupts persistency of the abstract algebraic semantics

33a ⟨*data types* 23a⟩+≡

```
settype         SubscriberSet is BaseId
set             SubscriberSet
elements        BaseId
endtype

type            SubscriberSetOpns is SubscriberSet
opns            pick : SubscriberSet -> BaseId
eqns forall b : BaseId, s : SubscriberSet
ofsort BaseId
        pick(insert_C(b,s)) = b ;
endtype
```

Defines:
`SubscriberSet`, used in chunks 38 and 69a.
Uses `BaseId` 23a.

Observers This is variable O of [Euz97a], ignored in the protocol and therefore not specified here.

Call for comments This is table C of [Euz97a]. A `pick` operation is added, as on `SubscriberSet`.

33b ⟨*data types* 23a⟩+≡

```
recordtype      CfcEntry is BaseId, Id, Request, NaturalNumber
record          cfc : CfcEntry
fields          id : Id
                sender : BaseId
                senderid : Id
                req : Request
                count : Nat
endtype

tabletype       CfcTbl is CfcEntry
table           CfcTbl
elements        CfcEntry
key             id : Id
endtype

type            CfcTblOpns is CfcTbl
opns            pick            : CfcTbl -> CfcEntry
                searchrequest   : BaseId, Id, CfcTbl -> Bool
                getrequest      : BaseId, Id, CfcTbl -> CfcEntry
                incallcount     : CfcTbl -> CfcTbl
```

```
eqns forall     e : CfcEntry,
                t : CfcTbl,
                b : BaseId,
                m : Id
ofsort CfcEntry
        pick(insert_C(e, t)) = e ;
ofsort Bool
        searchrequest(b, m, {}) = false ;
        sender(e) eq b, senderid(e) eq m =>
                searchrequest(b, m, insert_C(e, t)) = true ;
        searchrequest(b, m, insert_C(e, t)) =
                searchrequest(b, m, t) ;
ofsort CfcEntry
        sender(e) eq b, senderid(e) eq m =>
                getrequest(b, m, insert_C(e, t)) = e ;
        getrequest(b, m, insert_C(e, t)) =
                getrequest(b, m, t) ;
ofsort CfcTbl
        incallcount({}) = {} ;
        incallcount(insert_C(e, t)) =
                insert(set_count(succ(count(e)), e),
                        incallcount(t)) ;
endtype
```

Defines:
`CfcTbl`, used in chunks 38 and 69b.
Uses `BaseId` 23a, `Id` 23b, and `Request` 27.

Parent base This is either `none` (if the base has not successfully registered yet) or `some(b)` where `b` is the parent base id. Such a 'lifted' type is produced by the APERO `optiontype` declaration.

34 ⟨*data types* 23a⟩+≡

```
optiontype      ParentBase is BaseId
option          ParentBase
elements        BaseId
endtype

type            ParentBaseOpns is ParentBase
opns            the             : ParentBase -> BaseId
eqns forall b,b1 : BaseId
ofsort BaseId
        the(some(b)) = b ;
endtype
```

Defines:
`ParentBase`, used in chunks 37 and 38.
Uses `BaseId` 23a.

A.3.6 Miscellaneous Data Types

Numeric Operations The usual constants for small numbers, to avoid lengthy `succ(succ(...))` notations, and a predecessor function.

35a ⟨*data types* 23a⟩+≡

```
type            NatOpns         is NaturalNumber
opns            1,2,3,4,5,6,7,8,9 : -> Nat
                pred            : Nat -> Nat
eqns ofsort Nat
forall x : Nat
        1 = succ(0) ;
        2 = succ(1) ;
        3 = succ(2) ;
        4 = succ(3) ;
        5 = succ(4) ;
        6 = succ(5) ;
        7 = succ(6) ;
        8 = succ(7) ;
        9 = succ(8) ;
        pred(succ(x)) = x ;
endtype
```

Message Queues `PacketQueue` is used to hold the content of a queue of messages.

35b ⟨*data types* 23a⟩+≡

```
recordtype      Packet is Message, BaseId
record          packet : Packet
fields          sender : BaseId
                receiver : BaseId
                message : Message
endtype

listtype      PacketQueue is Packet
list          PacketQueue
elements      Packet
endtype
```

Defines:

`Packet`, never used.
`PacketQueue`, used in chunk 42c.
Uses `BaseId` 23a and `Message` 28b.

A.4 Global Structure

A Co_4 system is made up of a collection of *knowledge bases* (KBs) organized as a tree. The leaves are *individual bases* directly connected to a single human user. The other nodes are *group bases* whose content is the agreed common knowledge of their children bases. This structure is dynamic; new children may be added while the system is running.

The core of the behaviour part of the specification will be a pair of process definitions `IndividualKB` and `GroupKB`, modelling resp. individual and group bases. n this specification, we do not address the possibility that an individual may become a group or vice-versa.

Their behaviour will be a never-ending loop, where one 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.

In [Euz97a], $!n$ denotes the generation of a "brand new surrogate" n. We specify this using a seed value that is "incremented" after each use. these seeds are state variables NA (for request ids in A) and NC (for cfc ids in C, group bases only).

`InitIndividualKB [...] (b)` describes an initial base with identifier `b`, whereas `IndividualKB [...] (b,...)` is a base with all its state variables. The same applies for `InitGroupKB` and `GroupKB`.

`IdleIndividualKB [...] (b,G)` is an individual base with empty tables but possibly with a known group base. The same applies to `IdleGroupKB`, where the set of subscribed bases is also provided.

All these processes are declared with functionality `exit`, and can terminate at any moment between the application of two rules, provided that all tables holding transaction information are empty. The environments used for simulation and verification will also end with process termination; this will allow to differenciate between normal (exit) and abnormal (deadlock) sink states.

A.4.1 Individual bases

An individual base is a process with gate `user` to interact with the human user and `parentin`, `parentout` to communicate with its parent base: events on `parentin` are initiated by the parent base (input) whereas events no `parentout` are initiated by the base (output). In addition, a gate `signal` supports events used to monitor some internal conditions in the base. Events on those gates have the following structure:

```
user !BaseId !UserAction ?<args> ...
parentin ?BaseId !BaseId ?Message
parentout !BaseId !BaseId !Message
signal !BaseId !SignalVal !<args>...
```

37 ⟨*IndividualKB processes* 37⟩≡

```
process InitIndividualKB [user,parentin,parentout,signal]
  ( B : BaseId ) : exit :=
  IndividualKB [user,parentin,parentout,signal]
        (B,
        none of ParentBase,
        {} of SubmittedTbl, first of Id,
        {} of PendingTbl)
endproc

process IdleIndividualKB [user,parentin,parentout,signal]
  ( B : BaseId,
    G : ParentBase ) : exit :=
  IndividualKB [user,parentin,parentout,signal]
        (B,
        G,
        {} of SubmittedTbl, first of Id,
        {} of PendingTbl)
endproc

process IndividualKB [user,parentin,parentout,signal]
  ( B : BaseId,
    G : ParentBase,
    A : SubmittedTbl, NA : Id,
    P : PendingTbl)
  : exit :=

  [A = {} of SubmittedTbl] -> [P = {} of PendingTbl] -> exit
  []
  ⟨individual rules 44⟩

endproc
```

Defines:
  `IndividualKB`, used in chunks 39b, 45–47, 49, and 50.
  `InitIndividualKB`, used in chunks 74b, 76–81, and 86a.
Uses `BaseId` 23a, `Id` 23b, `ParentBase` 34, `PendingTbl` 31, and `SubmittedTbl` 30b.

A.4.2 Group bases

A group base communicates with its own parent via `parentin`, `parentout` and with its children via `childrenin`, `childrenout`. We also add a gate `signal` as in individual bases. Events have the following structure:

```
childrenin ?BaseId !BaseId ?Message
childrenout !BaseId !BaseId !Message
parentin ?BaseId !BaseId ?Message
parentout !BaseId !BaseId !Message
signal !BaseId !SignalVal !<args>...
```

38 ⟨*GroupKB processes* 38⟩≡

```
process InitGroupKB [childrenin,childrenout,parentin,parentout,signal]
  ( B : BaseId ) : exit :=
  GroupKB [childrenin,childrenout,parentin,parentout,signal]
        (B,
        none of ParentBase,
        {} of GrpSubmittedTbl, first of Id,
        {} of PendingTbl,
        {} of CfcTbl, first of Id,
        {} of SubscriberSet,
        {} of Knowledge)
endproc

process IdleGroupKB [childrenin,childrenout,parentin,parentout,signal]
  ( B : BaseId,
    G : Parentbase,
    S : SubscriberSet ) : exit :=
  GroupKB [childrenin,childrenout,parentin,parentout,signal]
        (B,
        G,
        {} of GrpSubmittedTbl, first of Id,
        {} of PendingTbl,
        {} of CfcTbl, first of Id,
        S,
        {} of Knowledge)
endproc

process GroupKB [childrenin,childrenout,parentin,parentout,signal]
  ( B : BaseId,
    G : Parentbase,
    A : GrpSubmittedTbl, NA : Id,
    P : PendingTbl,
    C : CfcTbl, NC : Id,
    S : SubscriberSet,
    K : Knowledge)
  : exit :=

    [A = {} of GrpSubmittedTbl] -> [P = {} of PendingTbl] ->
      [C = {} of CfcTbl] -> exit
```

```
  []
  ⟨group rules 51⟩

endproc
```

Defines:
  GroupKB, used in chunks 39b, 52–58, 60–65, 67, and 68.
  InitGroupKB, used in chunks 74b, 76–81, and 86a.
Uses BaseId 23a, CfcTbl 33b, GrpSubmittedTbl 30b, Id 23b, Knowledge 25b, ParentBase 34, PendingTbl 31, and SubscriberSet 33a.

A.4.3 Top Level structure

Abstract structure This section defines the top-level structure of an abtract specification of Co_4, i.e. one that would intend to formalize in all generality the behaviour of a Co_4 system without any concern about subsequent verification needs. It is *not* the specification that we have used (that one is defined below); to emphasize this fact, the description uses the conditional.

Since the structure of the base tree changes dynamically, we could not reflect it in the synchronization structure of the specification. Instead, we would let any child base communicate with any parent, and vice-versa, through a communication channel.

39a ⟨*abstract com channel* 39a⟩≡

```
process ComChannel [parentin,parentout,childrenin,childrenout] : noexit :=

  ⟨abstract com channel behaviour (never defined)⟩

endproc
```

where `ComChannel` would essentially perform `childrenin !from !to !msg` for every `parentout !from !to !msg` and conversely, *mutatis mutandis*, for `parentin` and `childrenout`.

We would then specify an unbounded number of concurrent individual and group bases, using a recursive parallel construct:

39b ⟨*abstract pool of KBs* 39b⟩≡

```
process KBPool [user,parentin,parentout,childrenin,childrenout]
(bs: BaseIdSet) : noexit :=

  choice b: BaseId [] [b notin bs] ->
  ( ( i; IndividualKB [user,parentin,parentout] (b,...)
      []
      i; GroupKB [childrenin,childrenout,parentin,parentout] (b,...)
    )
```

```
    |||
    KBPool [user,parentin,parentout,childrenin,childrenout] (insert(b,bs))
  )

endproc
```

Uses `BaseId` 23a, `GroupKB` 38, and `IndividualKB` 37.

The top level would synchronize this pool of bases through the communication channel:

40 ⟨*abstract specification* 40⟩≡

```
specification CO4System [user] : noexit

⟨data types 23a⟩

behaviour

  hide parentin,parentout,childrenin,childrenout in
  ( KBPool [user,parentin,parentout,childrenin,childrenout]
        ({} of BaseIdSet)
    |[parentin,parentout,childrenin,childrenout]|
    ComChannel [parentin,parentout,childrenin,childrenout]
  )

where

⟨processes (never defined)⟩
⟨abstract pool of KBs 39b⟩
⟨abstract com channel 39a⟩

endproc
```

Simplified structure The abstract specification described above is not usable as a basis for model checking, because it contains unbounded parallelism in the recursive definition `KBPool`. This is rejected by the model compiler. Furthermore, It describes any potential behaviour of any collection of bases, which is a far too general case to be handled by model-based verification.

Instead of this, we apply our verification tools to static (and small) combinations of individual and group bases, and we further restrict the explored scenarios by coupling the system with an environment producing only some chosen external events. By the nature of the CO_4 protocol, this is enough to constrain the possible scenarios throughout the whole hierarchy.

To make the inner workings of the protocol observable, we also do not hide the internal gates `childrenin`, `childrenout`, `userin` and `userout`.

The precise structure of the specification will vary from one experiment to another, but all variants share the same general structure: the global behaviour is a fixed hierarchy of interconnected bases, synchronized on gate `user` with a user environment that restricts the spectrum of possible behaviours. This whole structure is followed sequentially with a looping process, so that normal termination does not produce a deadlock state.

41 ⟨*specification* 41⟩≡

```
specification CO4System [user,⟨KB gates (never defined)⟩,signal,OK] : noexit

  ⟨data types 23a⟩

behaviour

(
  ( UserInput [user] [> exit )
  |[user]|
  KBHierarchy [user,⟨KB gates (never defined)⟩,signal]
)
>> Ring [OK]

where

  ⟨KBHierarchy process (never defined)⟩
  ⟨UserInput process (never defined)⟩

  ⟨IndividualKB processes 37⟩
  ⟨GroupKB processes 38⟩
  ⟨Queue processes 42c⟩
  ⟨Ring process 42a⟩

endspec
```

Uses `Ring` 42a.

The following chunks vary according to the scenario:

- ⟨*KBHierarchy process* (never defined)⟩ defines the hierarchy of bases as a process `KBHierarchy`,
- ⟨*KB gates* (never defined)⟩ is the list of gates between bases, and is thus derived from the interface of `KBHierarchy`,
- ⟨*UserInput process* (never defined)⟩ defines the user input scenario as a process `UserInput`.

The different hierarchies and scenarios are detailed in the last part of this document.

A.4.4 Normal Terminating State

In verification we have to limit ourselves to finite scenarios. In this case, we need to detect a difference between terminal states due to a deadlock in the system and those resulting from normal end of activity. For this purpose we replace the latter by a state where the system continuously "rings" a given action. This state is described by the following process `Ring`.

42a ⟨*Ring process* 42a⟩≡

```
process Ring [g] : noexit :=
  g; Ring [g]
endproc
```

Defines:
`Ring`, used in chunk 41.

A.4.5 Communication Queues

This section defines communication channels that are used to introduce asynchronism in the interactions between bases. It is assumed that the Co_4 protocol runs over a transport service that preserves uniqueness, integrity and order between messages, i.e. basically an idealized FIFO queue. However, queues are introduced only where absolutely needed, since they are a major source of asynchronism and therefore state space explosion — that is, we avoid queues as long as the resulting specification does not present deadlocks that are due to lack of asynchronism (as was the case in the fully synchronous version 1).

The basic process `Cell` defines a one-slot queue; it can be chained to form longer but still bounded queues.

42b ⟨*Cell processes* 42b⟩≡

```
process Cell [input,output] : exit :=
  exit
  []
  input ?b1 : BaseId ?b2 : BaseId ?msg : Message;
  output !b1 !b2 !msg;
  Cell [input,output]
endproc
```

Defines:
`Cell`, never used.
Uses `BaseId` 23a and `Message` 28b.

However bounded queues are not sufficient in general. The process `Queue` hereafter defines unbounded queues using a state variable.

42c ⟨*Queue processes* 42c⟩≡

```
process InitQueue [input,output] : exit :=
  Queue  [input,output] (<> of PacketQueue)
endproc

process Queue [input,output] (pktq : PacketQueue) : exit :=
  [pktq eq <>] -> exit
  []
  input ?b1 : BaseId ?b2 : BaseId ?msg : Message;
  Queue [input,output] (pktq + packet(b1,b2,msg))
  []
  [pktq ne <>] ->
  output !sender(first(pktq)) !receiver(first(pktq)) !message(first(pktq)) ;
  Queue [input,output] (butfirst(pktq))
endproc
```

Defines:
`InitBufGroupKB`, never used.
`InitQueue`, used in chunks 74b, 77–81, 83b, and 86a.
`Queue`, never used.
Uses `BaseId` 23a, `Message` 28b, and `PacketQueue` 35b.

Experience shows that introducing unbounded queues in one flow direction suffices and that upward queues (i.e. from children to parent base) produce the smaller models.

A.5 Rules

This section contains the part of the specification that corresponds to the rules in [Euz97a]. These rules have the following general form[8]:

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

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).

Each such rule is modelled as a branch of a choice, along the following template:

43 ⟨*sample rule* 43⟩≡

```
process KB [gatein,gateout] (V:state ...) : noexit :=
gatein ?s:BaseId !b ?msg:Message;
( ...
```

[8]The fully general form allows zero or several variable updates and output messages.

```
    []
    [is_m1(msg)] ->
    ( let x:data = content(msg) in
        [P] -> gateout !b !r !m2(z); KB [gatein,gateout] (y ...)
    )
    []
    ...
  )
  endproc
```

Uses `BaseId` 23a and `Message` 28b.

This is merely a general sketch; the smaller details will vary from rule to rule. In the case of actions triggered by the user, the first event offer is replaced by an interaction with the user.

Only valid user initiatives are accepted (e.g. voting is allowed only on new cfcs, submissions etc. are not allowed if the base has no parent). On the other hand, internal messages of the protocol are accepted first and the validity of their content is checked afterwards. The occurrence of error conditions (e.g. entries not found in tables or messages received from wrong bases) produces a local deadlock of the concerned base. The reasoning that supports this way of doing things is that we want to check that the protocol is free from invalid messages, whereas we reject a priori invalid user requests.

A.5.1 Rules for Individual bases

There are two kinds of rules for individual bases: *user initiatives*, triggered by interactions with the user, and *automatic rules*, triggered by a CO_4 message from the parent base.

44 ⟨*individual rules* 44⟩≡

```
  ⟨user register rules 45a⟩
  []
  ⟨user evaluate rules 45b⟩
  []
  ⟨user achieve rules 46a⟩
  []
  ⟨user forward rules 46b⟩
  []
  ⟨user deny rules 46c⟩
  []
  ⟨user reply rules 47a⟩
  []
  parentin ?from : BaseId !B ?msg : Message;
  ( ⟨user notify rules 47b⟩
    []
    ⟨user askall rules 49a⟩
```

```
    []
    ⟨user tell rules 49b⟩
    []
    ⟨user error rules 49c⟩
    []
    ⟨user poolnotify rules 50a⟩
    []
    ⟨user pooldeny rules 50b⟩
  )
```

Uses `BaseId` 23a and `Message` 28b.

In rules [evaluate], [achieve], [forward] and [deny], the condition that $G \neq \emptyset$ has been added, i.e. the base must have registered beforehand. A—likely unintended—consequence of this is that it is not possible to deny a register request.

[register]

45a ⟨*user register rules* 45a⟩≡

```
  [G = none] -> [A = {} of SubmittedTbl] ->
  user !B !doregister ?b1 : BaseId;
  parentout !B !b1 !register(NA,b1);
  IndividualKB [user,parentin,parentout,signal]
          (B,
          G,
          insert(submitted(NA,reqregister(b1)),A),next(NA),
          P)
```

Uses `BaseId` 23a, `IndividualKB` 37, and `SubmittedTbl` 30b.

[evaluate]

45b ⟨*user evaluate rules* 45b⟩≡

```
  [issome(G)] ->
  user !B !doevaluate ?p1 : Proposal;
  parentout !B !the(G) !evaluate(NA,p1);
  IndividualKB [user,parentin,parentout,signal]
          (B,
          G,
          insert(submitted(NA,reqevaluate(p1)),A),next(NA),
          P)
```

Uses `IndividualKB` 37 and `Proposal` 25b.

[achieve]

46a ⟨*user achieve rules* 46a⟩≡

```
[issome(G)] ->
user !B !doachieve ?p1 : Proposal;
parentout !B !the(G) !achieve(NA,p1);
IndividualKB [user,parentin,parentout,signal]
        (B,
        G,
        insert(submitted(NA,reqachieve(p1)),A),next(NA),
        P)
```

Uses `IndividualKB` 37 and `Proposal` 25b.

[forward]

46b ⟨*user forward rules* 46b⟩≡

```
[issome(G)] ->
user !B !doforward ?r1 : Request;
parentout !B !the(G) !forward(NA,r1);
IndividualKB [user,parentin,parentout,signal]
      (B,
      G,
      insert(submitted(NA,reqforward(r1)),A),next(NA),
      P)
```

Uses `IndividualKB` 37 and `Request` 27.

[deny]

46c ⟨*user deny rules* 46c⟩≡

```
[issome(G)] ->
( choice m1 : Id [] [m1 isin A] ->
  user !B !dodeny !req(get(m1,A));
  parentout !B !the(G) !deny(m1);
  IndividualKB [user,parentin,parentout,signal]
        (B,
        G,
        remove(m1,A),NA,
        P)
)
```

Uses `Id` 23b and `IndividualKB` 37.

[accept], [reject], [challenge] Note that only *achieve* requests can be challenged. The possibility to challenge $forward^*(achieve(\ldots))$ requests is still under analysis and not covered by this specification.

47a ⟨*user reply rules* 47a⟩≡

```
[issome(G)] ->
( choice n1 : Id []
  [n1 isin P] ->  [status(get(n1,P)) = new] ->
  ( let r1 : Request = req(get(n1,P)) in
    user !B !doaccept !r1;
    parentout !B !the(G) !reply(n1,acceptx);
    IndividualKB [user,parentin,parentout,signal]
        (B,
        G,
        A,NA,
        insert(set_status(accepted,get(n1,P)),P))
    []
    user !B !doreject !r1;
    parentout !B !the(G) !reply(n1,reject);
    IndividualKB [user,parentin,parentout,signal]
        (B,
        G,
        A,NA,
        insert(set_status(rejected,get(n1,P)),P))
    []
    [is_reqachieve(r1)] ->
    (*** should apply to forward*(achieve(...)) too ***)
    user !B !dochallenge !r1 ?p1 : Proposal;
    parentout !B !the(G) !reply(n1,challenge(NA,p1));
    IndividualKB [user,parentin,parentout,signal]
        (B,
        G,
        insert(submitted(NA,reqchallenge(p1)),A),next(NA),
        insert(set_status(challenged,get(n1,P)),P))
  )
)
```

Uses `Id` 23b, `IndividualKB` 37, `Proposal` 25b, and `Request` 27.

Now comes the automatic part of individual bases.

(notify-register)

47b ⟨*user notify rules* 47b⟩≡

```
[is_notify(msg)] ->
( let   m1 : Id = inreplyto(msg),
```

```
        a1 : Answer = content(msg) in
  ( [m1 isin A] ->
    ( [is_reqregister(req(get(m1,A)))] ->
      [content(req(get(m1,A))) of BaseId = from] ->
      ( [isnone(G)] ->
        ( [is_acceptx(a1)] ->
          IndividualKB [user,parentin,parentout,signal]
                (B,
                some(from),
                remove(m1,A),NA,
                P)
          []
          [not(is_acceptx(a1))] ->
          IndividualKB [user,parentin,parentout,signal]
                (B,
                G,
                remove(m1,A),NA,
                P)
        )
        []
        [issome(G)] ->
        IndividualKB [user,parentin,parentout,signal]
                (B,
                G,
                remove(m1,A),NA,
                P)
      )
      []
      [not(is_reqregister(req(get(m1,A))))] ->
      IndividualKB [user,parentin,parentout,signal]
              (B,
              G,
              remove(m1,A),NA,
              P)
    )
    []
    [m1 notin A] ->
    signal !B !signotinA !msg;
    IndividualKB [user,parentin,parentout,signal]
        (B,
        G,
        A,NA,
        P)
  )
)
```

Uses `Answer` 28a, `BaseId` 23a, `Id` 23b, and `IndividualKB` 37.

(store-cfc)

49a ⟨*user askall rules* 49a⟩≡

```
[is_askall(msg)] ->
( [G eq some(from)] ->
  ( let n1 : Id = replywith(msg),
        r1 : Request = content(msg) in
    IndividualKB [user,parentin,parentout,signal]
        (B,
        G,
        A,NA,
        insert(pending(n1,r1,new),P))
  )
)
```

Uses `Id` 23b, `IndividualKB` 37, and `Request` 27.

(store-proposal)

49b ⟨*user tell rules* 49b⟩≡

```
[is_tell(msg)] ->
( [G eq some(from)] ->
  (*** handling of knowledge in user KB: unspecified ***)
  IndividualKB [user,parentin,parentout,signal]
    (B,
    G,
    A,NA,
    P)
)
```

Uses `IndividualKB` 37.

(receive-error)

49c ⟨*user error rules* 49c⟩≡

```
[is_error(msg)] ->
( [G eq some(from)] ->
  ( let m1 : Id = inreplyto(msg) in
    ( [m1 isin A] ->
      IndividualKB [user,parentin,parentout,signal]
        (B,
```

```
          G,
          remove(M1,A),NA,
          P)
        []
        [m1 notin A] ->
        signal !B !signotinA !msg;
        IndividualKB [user,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            P)
      )
    )
  )
```

Uses `Id` 23b and `IndividualKB` 37.

(handle-pool-notify)

50a ⟨*user poolnotify rules* 50a⟩≡

```
  [is_poolnotify(msg)] ->
  ( [G eq some(from)] ->
    ( let n1 : Id = inreplyto(msg) in
      ( [n1 isin P] ->
        IndividualKB [user,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          remove(n1,P))
        []
        [n1 notin P] ->
        signal !B !signotinP !msg;
        IndividualKB [user,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            P)
      )
    )
  )
```

Uses `Id` 23b and `IndividualKB` 37.

(cancel-cfc)

50b ⟨*user pooldeny rules* 50b⟩≡

```
[is_pooldeny(msg)] ->
( [G eq some(from)] ->
  ( let n1 : Id = inreplyto(msg) in
    ( [n1 isin P] ->
      IndividualKB [user,parentin,parentout,signal]
        (B,
        G,
        A,NA,
        remove(n1,P))
      []
      [n1 notin P] ->
      signal !B !signotinP !msg;
      IndividualKB [user,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P)
    )
  )
)
```

Uses `Id` 23b and `IndividualKB` 37.

A.5.2 Rules for Group bases

We distinguish between rules triggered by messages coming from a child or a parent. Each half is given as a choice guarded by a corresponding event offer.

51 ⟨*group rules* 51⟩≡

```
childrenin ?from : BaseId !B ?msg : Message;
( ⟨group register rules 52⟩
  []
  ⟨group evaluate rules 53a⟩
  []
  ⟨group achieve rules 53b⟩
  []
  ⟨group forward rules 54⟩
  []
  ⟨group deny rules 55⟩
  []
  ⟨group reply rules 56⟩
)
[]
```

```
  parentin ?from : BaseId !B ?msg : Message;
  ( ⟨group notify rules 63⟩
    []
    ⟨group askall rules 64⟩
    []
    ⟨group error rules 65a⟩
    []
    ⟨group poolnotify rules 67⟩
    []
    ⟨group pooldeny rules 65b⟩
    []
    ⟨group tell rules 68⟩
  )
```

Uses `BaseId` 23a and `Message` 28b.

(cfc-register)
(reply-register)

52 ⟨*group register rules* 52⟩≡

```
  [is_register(msg)] ->
  ( let   b2 : BaseId = content(msg),
          m1 : Id = replywith(msg) in
    let   req : Request = reqregister(b2) in
    [b2 = B] ->
    ( [S ne {}] ->
      ( Broadcast [childrenout] (B, askall(NC,req), S)
        >>
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
              (B,
              G,
              A,NA,
              P,
              insert(cfc(NC,from,m1,req,card(S)),C), next(NC),
              S,
              K)
      )
      []
      [S eq {}] ->
      childrenout !B !from !notify(m1, acceptx);
      childrenout !B !from !tell(assert(K));
      signal !B !sigregistered !from;
      GroupKB [childrenin,childrenout,parentin,parentout,signal]
              (B,
              G,
```

```
              A,NA,
              P,
              C,NC,
              insert(from,S),
              K)
    )
  )
```

Uses BaseId 23a, Broadcast 69a, GroupKB 38, Id 23b, and Request 27.

(eval-reply)

53a ⟨*group evaluate rules* 53a⟩≡

```
  [is_evaluate(msg)] ->
  ( let   p1 : Proposal = content(msg),
          m1 : Id = replywith(msg) in
    childrenout !B !from !notify(m1, answer(query(K,p1)));
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P,
          C,NC,
          S,
          K)
  )
```

Uses GroupKB 38, Id 23b, and Proposal 25b.

(cfc-achieve)
(error-achieve)

53b ⟨*group achieve rules* 53b⟩≡

```
  [is_achieve(msg)] ->
  ( let   p1 : Proposal = content(msg),
          m1 : Id = replywith(msg) in
    let   req : Request = reqachieve(p1) in
    ( [query(K,p1)] ->
      ( Broadcast [childrenout] (B, askall(NC,req), S)
        >>
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
              (B,
              G,
              A,NA,
```

```
            P,
            insert(cfc(NC,from,m1,req,card(S)),C),next(NC),
            S,
            K)
      )
      []
      [not(query(K,p1))] ->
      ( childrenout !B !from !error(m1);
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            P,
            C,NC,
            S,
            K)
      )
    )
  )
```

Uses Broadcast 69a, GroupKB 38, Id 23b, Proposal 25b, and Request 27.

(cfc-forward)
(error-forward)

54 ⟨*group forward rules* 54⟩≡

```
  [is_forward(msg)] ->
  ( let   r1 : Request = content(msg),
          m1 : Id = replywith(msg) in
    let   req : Request = reqforward(r1) in
    ( [issome(G) or (isnone(G) and is_reqregister(r1))] ->
      ( Broadcast [childrenout] (B, askall(NC,req), S)
        »
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            P,
            insert(cfc(NC,from,m1,req,card(S)),C),next(NC),
            S,
            K)
      )
      []
      [isnone(G) and not(is_reqregister(r1))] ->
      ( childrenout !B !from !error(m1);
```

```
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
              (B,
              G,
              A,NA,
              P,
              C,NC,
              S,
              K)
      )
    )
  )
```

Uses `Broadcast` 69a, `GroupKB` 38, `Id` 23b, and `Request` 27.

(deny-reply)

55 ⟨*group deny rules* 55⟩≡

```
  [is_deny(msg)] ->
  ( let   m1 : Id = inreplyto(msg) in
    [searchrequest(from,m1,C)] ->
    ( let n1 : Id = id(getrequest(from,m1,C)) in
      ( Broadcast [childrenout] (B, pooldeny(n1), S)
        >>
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P,
          remove(n1,C),NC,
          S,
          K)
      )
    )
    []
    [not(searchrequest(from,m1,C))] ->
    signal !B !signotinC !msg;
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P,
          C,NC,
          S,
          K)
```

```
)
```

Uses `Broadcast` 69a, `GroupKB` 38, and `Id` 23b.

The following piece gathers all rules pertaining to the reception of replies in a group base. The treatment of *reply* is rather complex and involves a dozen rules, so we further decompose it in several chunks.

56 ⟨*group reply rules* 56⟩≡

```
[is_reply(msg)] ->
( let   n1 : Id = inreplyto(msg),
        a1 : Answer = content(msg) in
  [n1 isin C] ->
  ( let entry : CfcEntry = get(n1,C) in
    let x1 : Nat = count(entry),
        r1 : Request = req(entry),
        b2 : BaseId = sender(entry),
        m2 : Id = senderid(entry) in
    ( [is_acceptx(a1)] ->
      ( [x1 = 1] ->
        (
          ⟨group accept register rules 57⟩
          []
          ⟨group accept achieve rules 58a⟩
          []
          ⟨group accept forward rules 58b⟩
          []
          ⟨group accept tell rules 60⟩
        )
        []
        [x1 gt 1] ->
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            P,
            insert(set_count(pred(x1),entry),C),NC,
            S,
            K)
      )
      []
      [is_reject(a1)] ->
      (
        ⟨group reject rules 61⟩
      )
      []
```

```
        [is_challenge(a1)] ->
        (
          ⟨group challenge rules 62⟩
        )
      )
    )
    []
    [n1 notin C] ->
    signal !B !signotinC !msg;
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P,
          C,NC,
          S,
          K)
  )
```

Uses `Answer` 28a, `BaseId` 23a, `GroupKB` 38, `Id` 23b, and `Request` 27.

(accept-register)

57 ⟨*group accept register rules* 57⟩≡

```
  [is_reqregister(r1)] ->
  childrenout !B !b2 !notify(m2, acceptx);
  childrenout !B !b2 !tell(assert(K));
  ( Broadcast [childrenout] (B, poolnotify(n1,acceptx), S)
    >>
    EnumerateAskall [childrenout] (B, remove(n1,C), b2)
    >>
    signal !B !sigregistered !b2;
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P,
          incallcount(remove(n1,C)),NC,
          insert(b2,S),
          K)
  )
```

Uses `Broadcast` 69a, `EnumerateAskall` 69b, and `GroupKB` 38.

(accept-achieve)

58a ⟨*group accept achieve rules* 58a⟩≡

```
[is_reqachieve(r1)] ->
childrenout !B !b2 !notify(m2, acceptx);
( let p1 : Proposal = content(r1) in
  ( Broadcast [childrenout] (B, poolnotify(n1,acceptx), S)
    »
    Broadcast [childrenout] (B, tell(p1), S)
    »
    signal !B !sigstored !p1 !query(K,p1);
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
        (B,
        G,
        A,NA,
        P,
        remove(n1,C),NC,
        S,
        K+p1)
  )
)
```

Uses `Broadcast` 69a, `GroupKB` 38, and `Proposal` 25b.

(accept-forward)
(accept-forward-register)

58b ⟨*group accept forward rules* 58b⟩≡

```
[is_reqforward(r1)] ->
( [not(G eq some(b2))] ->
  ( let r2 : Request = content(r1) in
    Broadcast [childrenout] (B, poolnotify(n1,acceptx), S)
    »
    ( [is_reqregister(r2)] ->
      ( let b3 : BaseId = content(r2) in
        parentout !B !b3 !register(NA,b3);
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          insert(submitted(NA,r2,b2,m2),A),next(NA),
          P,
          remove(n1,C),NC,
          S,
          K)
      )
```

```
    []
    [is_reqevaluate(r2)] -> [issome(G)] ->
    parentout !B !the(G) !evaluate(NA,content(r2));
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
        (B,
        G,
        insert(submitted(NA,r2,b2,m2),A),next(NA),
        P,
        remove(n1,C),NC,
        S,
        K)
    []
    [is_reqachieve(r2)] -> [issome(G)] ->
    parentout !B !the(G) !achieve(NA,content(r2));
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
        (B,
        G,
        insert(submitted(NA,r2,b2,m2),A),next(NA),
        P,
        remove(n1,C),NC,
        S,
        K)
    []
    [is_reqforward(r2)] -> [issome(G)] ->
    parentout !B !the(G) !forward(NA,content(r2));
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
        (B,
        G,
        insert(submitted(NA,r2,b2,m2),A),next(NA),
        P,
        remove(n1,C),NC,
        S,
        K)
  )
)
[]
[G eq some(b2)] ->
( [m2 isin P] -> [status(get(m2,P)) = new] ->
  parentout !B !the(G) !reply(m2,acceptx);
  ( Broadcast [childrenout] (B, poolnotify(n1,acceptx), S)
    >>
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
      (B,
      G,
      A,NA,
      insert(set_status(accepted,get(m2,P)),P),
```

```
          remove(n1,C),NC,
          S,
          K)
      )
      []
      [m2 notin P] ->
      signal !B !signotinP !msg;
      ( Broadcast [childrenout] (B, poolnotify(n1,acceptx), S)
        »
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P,
          remove(n1,C),NC,
          S,
          K)
      )
    )
  )
```

Uses `BaseId` 23a, `Broadcast` 69a, `GroupKB` 38, and `Request` 27.

(broadcast-tell)

60 ⟨*group accept tell rules* 60⟩≡

```
  [is_reqtell(r1)] -> [G eq some(b2)] ->
  ( let p1 : Proposal = content(r1) in
    ( Broadcast [childrenout] (B, poolnotify(n1,acceptx), S)
      »
      Broadcast [childrenout] (B, tell(p1), S)
      »
      signal !B !sigstored !p1 !query(K,p1);
      GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P,
          remove(n1,C),NC,
          S,
          K+p1)
    )
  )
```

Uses `Broadcast` 69a, `GroupKB` 38, and `Proposal` 25b.

(reject-reply)

61 ⟨*group reject rules* 61⟩≡

```
[not(G eq some(b2))] ->
childrenout !B !b2 !notify(m2, a1);
( Broadcast [childrenout] (B, pooldeny(n1), S)
  ≫
  GroupKB [childrenin,childrenout,parentin,parentout,signal]
        (B,
        G,
        A,NA,
        P,
        remove(n1,C),NC,
        S,
        K)
)
[]
[G eq some(b2)] ->
( [is_reqforward(r1)] ->
  ( [m2 isin P] -> [status(get(m2,P)) = new] ->
    parentout !B !the(G) !reply(m2,reject);
    ( Broadcast [childrenout] (B, pooldeny(n1), S)
      ≫
      GroupKB [childrenin,childrenout,parentin,parentout,signal]
        (B,
        G,
        A,NA,
        insert(set_status(rejected,get(m2,P)),P),
        remove(n1,C),NC,
        S,
        K)
    )
    []
    [m2 notin P] ->
    signal !B !signotinP !msg;
    ( Broadcast [childrenout] (B, pooldeny(n1), S)
      ≫
      GroupKB [childrenin,childrenout,parentin,parentout,signal]
        (B,
        G,
        A,NA,
        P,
        remove(n1,C),NC,
        S,
        K)
    )
```

```
    )
    []
    [is_reqtell(r1)] ->
    ( Broadcast [childrenout] (B, pooldeny(n1), S)
      ≫
      GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P,
          remove(n1,C),NC,
          S,
          K)
    )
  )
```

Uses `Broadcast` 69a and `GroupKB` 38.

(challenge-reply)

62 ⟨*group challenge rules* 62⟩≡

```
  [is_reqachieve(r1)] ->
  (*** should apply to forward*(achieve(...)) too ***)
  ( [not(G eq some(b2))] ->
    ( let m1 : Id = replywith(a1),
              p1 : Proposal = content(a1),
              p2 : Proposal = content(r1) in
      let r2 : Request = reqachieve(p2*p1) in
      childrenout !B !b2 !notify(m2, a1);
      ( Broadcast [childrenout] (B, pooldeny(n1), S)
        ≫
        Broadcast [childrenout] (B, askall(NC,r2), S)
        ≫
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            P,
            insert(       cfc(NC,from,m1,r2,card(S)),
                    remove(n1,C)),
            next(NC),
            S,
            K)
      )
    )
```

```
    []
    [G eq some(b2)] ->
    ( stop
      (*** should not be reached ***)
    )
  )
```

Uses `Broadcast` 69a, `GroupKB` 38, `Id` 23b, `Proposal` 25b, and `Request` 27.

Now comes the part of group bases concerned with messages coming from the parent group.

(broadcast-notify)
(broadcast-notify-register)

63 ⟨*group notify rules* 63⟩≡

```
  [is_notify(msg)] ->
  ( let   m1 : Id = inreplyto(msg),
          a1 : Answer = content(msg) in
    [m1 isin A] ->
    ( let entry : GrpSubmittedEntry = get(m1,A) in
      let r1 : Request = req(entry),
          b2 : BaseId = sender(entry),
          m2 : Id = senderid(entry) in
      childrenout !B !b2 !notify(m2,a1);
      ( [is_reqregister(r1)] -> [content(r1) of BaseId = from] ->
        ( [is_acceptx(a1)] ->
          GroupKB [childrenin,childrenout,parentin,parentout,signal]
                (B,
                some(from),
                remove(m1,A),NA,
                P,
                C,NC,
                S,
                K)
          []
          [not(is_acceptx(a1))] ->
          GroupKB [childrenin,childrenout,parentin,parentout,signal]
                (B,
                G,
                remove(m1,A),NA,
                P,
                C,NC,
                S,
                K)
        )
```

```
        []
        [not(is_reqregister(r1))] ->
        [G eq some(from)] ->
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
                (B,
                G,
                remove(m1,A),NA,
                P,
                C,NC,
                S,
                K)
      )
    )
    []
    [m1 notin A] ->
    signal !B !signotinA !msg;
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          P,
          C,NC,
          S,
          K)
  )
```

Uses `Answer` 28a, `BaseId` 23a, `GroupKB` 38, `Id` 23b, and `Request` 27.

(broadcast-cfc)

64 ⟨*group askall rules* 64⟩≡

```
  [is_askall(msg)] ->
  ( [G eq some(from)] ->
    ( let n1 : Id = replywith(msg),
          r1 : Request = content(msg) in
      Broadcast [childrenout] (B, askall(NC,reqforward(r1)), S)
      ≫
      GroupKB [childrenin,childrenout,parentin,parentout,signal]
          (B,
          G,
          A,NA,
          insert(pending(n1,r1,new),P),
          insert(cfc(NC,from,n1,reqforward(r1),card(S)),C),next(NC),
          S,
          K)
```

```
    )
  )
```

Uses `Broadcast` 69a, `GroupKB` 38, `Id` 23b, and `Request` 27.

(broadcast-error)

65a ⟨*group error rules* 65a⟩≡

```
  [is_error(msg)] ->
  ( [G eq some(from)] ->
    ( let m1 : Id = inreplyto(msg) in
      [m1 isin A] ->
      ( let entry : GrpSubmittedEntry = get(m1,A) in
        let b2 : BaseId = sender(entry),
            m2 : Id = senderid(entry) in
        childrenout !B !b2 !error(m2);
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
                (B,
                G,
                remove(m1,A),NA,
                P,
                C,NC,
                S,
                K)
      )
      []
      [m1 notin A] ->
      signal !B !signotinA !msg;
      GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            P,
            C,NC,
            S,
            K)
    )
  )
```

Uses `BaseId` 23a, `GroupKB` 38, and `Id` 23b.

(broadcast-deny)

65b ⟨*group pooldeny rules* 65b⟩≡

```
[is_pooldeny(msg)] ->
( [G eq some(from)] ->
  ( let n1 : Id = inreplyto(msg) in
    [n1 isin P] ->
    ( [searchrequest(from,n1,C)] ->
      ( let n2 : Id = id(getrequest(from,n1,C)) in
        ( Broadcast [childrenout] (B, pooldeny(n2), S)
          ≫
          GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            remove(n1,P),
            remove(n2,C),NC,
            S,
            K)
        )
      )
      []
      [not(searchrequest(from,n1,C))] ->
      GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            remove(n1,P),
            C,NC,
            S,
            K)
    )
    []
    [n1 notin P] ->
    signal !B !signotinP !msg;
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            P,
            C,NC,
            S,
            K)
  )
)
```

Uses `Broadcast` 69a, `GroupKB` 38, and `Id` 23b.

(broadcast-poolnotify)

67 ⟨*group poolnotify rules* 67⟩≡

```
[is_poolnotify(msg)] ->
( [G eq some(from)] ->
  ( let n1 : Id = inreplyto(msg) in
    [n1 isin P] ->
    ( [searchrequest(from,n1,C)] ->
      ( let n2 : Id = id(getrequest(from,n1,C)) in
        ( Broadcast [childrenout] (B, poolnotify(n2,acceptx), S)
          >>
          GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            remove(n1,P),
            remove(n2,C),NC,
            S,
            K)
        )
      )
      []
      [not(searchrequest(from,n1,C))] ->
      GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            remove(n1,P),
            C,NC,
            S,
            K)
    )
    []
    [n1 notin P] ->
    signal !B !signotinP !msg;
    GroupKB [childrenin,childrenout,parentin,parentout,signal]
            (B,
            G,
            A,NA,
            P,
            C,NC,
            S,
            K)
  )
)
```

Uses Broadcast 69a, GroupKB 38, and Id 23b.

(log-tell)
(cfc-tell)

68 ⟨*group tell rules* 68⟩≡

```
[is_tell(msg)] ->
( [G eq some(from)] ->
  ( let p1 : Proposal = content(msg) in
    let req : Request = reqtell(p1) in
    ( [query(K,p1)] ->
      ( Broadcast [childrenout] (B, askall(NC,req), S)
        »
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
              (B,
              G,
              A,NA,
              P,
              insert(cfc(NC,from,first,req,card(S)),C),next(NC),
              S,
              K)
      )
      []
      [not(query(K,p1))] ->
      ( signal !B !signotconsistent !p1;
        GroupKB [childrenin,childrenout,parentin,parentout,signal]
              (B,
              G,
              A,NA,
              P,
              C,NC,
              S,
              K)
      )
    )
  )
)
```

Uses Broadcast 69a, GroupKB 38, Proposal 25b, and Request 27.

A.5.3 Auxiliary Process Definitions

This section defines two auxiliary processes that iteratively produce several messages:

- Broadcast sends a message to a set of (subscriber) bases.

- **EnumerateAskall** sends a list of pending cfcs to a (newly registered) base.

69a ⟨*GroupKB processes* 38⟩+≡

```
process Broadcast [out]
        ( B : BaseId,
          msg : Message,
          S : SubscriberSet ) : exit :=
[S eq {}] -> exit
[]
[S ne {}] ->
( let b1 : BaseId = pick(S) in
  out !B !b1 !msg;
  Broadcast [out] (B, msg, remove(b1,S))
)
endproc
```

Defines:
`Broadcast`, used in chunks 52–55, 57, 58, 60–62, 64, 65b, 67, and 68.
Uses `BaseId` 23a, `Message` 28b, and `SubscriberSet` 33a.

69b ⟨*GroupKB processes* 38⟩+≡

```
process EnumerateAskall [childrenout]
        ( B : BaseId,
          C : CfcTbl,
          b1 : BaseId ) : exit :=
[C eq {}] -> exit
[]
[C ne {}] ->
( let entry : CfcEntry = pick(C) in
  childrenout !B !b1 !askall(id(entry),req(entry));
  EnumerateAskall [childrenout] (B, remove(id(entry), C), b1)
)
endproc
```

Defines:
`EnumerateAskall`, used in chunk 57.
Uses `BaseId` 23a and `CfcTbl` 33b.

A.6 System scenarios

This section contains the different system scenarios used to study different aspects of the system.

A.6.1 User Environment processes

The behaviours described in this section define particular combinations of user actions on individual bases. These behaviours need not be finite: the CADP toolset provides tools that generate the model of the specification incrementally, driven by some exploration strategy. It is therefore possible to achieve validation results on a finite subpart of an infinite LTS. Typically, the tool EXHIBITOR is used to explore the LTS according to a given trace specification (e.g. all traces that contain only one *achieve* request). Nonetheless, it is still crucial to reduce the state space as much as possible: if the model is too large or infinite, the tools will take more time to produce results and will run forever if no result is to be found.

Two variants are defined for each kind of behaviour: `User<xyz>` applies to a single user, `AllUser<xyz>` applies concurrently to all users. In the case where all users are to exhibit the same kind of behaviour, the latter should replace concurrent instances of the former, producing less parallelism and thus better response time.

User Replies Unlimited number of actions that pertain to an existing conversation — accept, reject or deny pending proposals.

70a ⟨*UserReply process* 70a⟩≡

```
process UserReply [user]
        (B : BaseId) : noexit :=
user !B !doaccept ?r1 : Request;
UserReply [user] (B)
[]
user !B !doreject ?r1 : Request;
UserReply [user] (B)
[]
user !B !dodeny ?r1 : Request;
UserReply [user] (B)
endproc
```

Defines:
`UserReply`, used in chunk 85.
Uses `BaseId` 23a and `Request` 27.

70b ⟨*AllUserReply process* 70b⟩≡

```
process AllUserReply [user] : noexit :=
user ? b1 : BaseId !doaccept ?r1 : Request;
AllUserReply [user]
[]
user ? b1 : BaseId !doreject ?r1 : Request;
AllUserReply [user]
[]
user ? b1 : BaseId !dodeny ?r1 : Request;
```

```
AllUserReply [user]
endproc
```

Defines:
`AllUserReply`, used in chunks 75–77, 79c, 82b, and 84b.
Uses `BaseId` 23a and `Request` 27.

User Accept Replies Unlimited number of accept on pending proposals.

71a ⟨*UserAccept process* 71a⟩≡

```
process UserAccept [user]
        (B : BaseId) : noexit :=
user !B !doaccept ?r1 : Request;
UserAccept [user] (B)
endproc
```

Defines:
`UserAccept`, never used.
Uses `BaseId` 23a and `Request` 27.

71b ⟨*AllUserAccept process* 71b⟩≡

```
process AllUserAccept [user] : noexit :=
user ? b1 : BaseId !doaccept ?r1 : Request;
AllUserAccept [user]
endproc
```

Defines:
`AllUserAccept`, used in chunks 83a and 86c.
Uses `BaseId` 23a and `Request` 27.

General User Input `UserGeneral` provides a large range of user input actions, repeatedly and in any order. It does not perform *register* requests; these are supposed to be offered separately and only once.

`UserGeneral` produces a huge branching factor and thus a very rapid state blow-up; it is intended for use in interactive simulations of the full spectrum of Co_4's features.

71c ⟨*UserGeneral process* 71c⟩≡

```
process UserGeneral [user] (B : BaseId) : noexit :=
let p1 : Proposal = assert(black) + <>,
    p2 : Proposal = assert(white) + <>
in
  user !B !doevaluate !p1; UserGeneral [user] (B)
  []
```

```
    user !B !doevaluate !p2; UserGeneral [user] (B)
    []
    user !B !doachieve !p1; UserGeneral [user] (B)
    []
    user !B !doachieve !p2; UserGeneral [user] (B)
    []
    user !B !dodeny ?r1 : Request;  UserGeneral [user] (B)
    []
    user !B !doaccept ?r1 : Request;  UserGeneral [user] (B)
    []
    user !B !doreject ?r1 : Request;  UserGeneral [user] (B)
    []
    user !B !dochallenge !reqachieve(p1) !p2; UserGeneral [user] (B)
    []
    user !B !dochallenge !reqachieve(p2) !p1; UserGeneral [user] (B)
    []
    user !B !doforward !reqachieve(p1); UserGeneral [user] (B)
    []
    user !B !doforward !reqachieve(p2); UserGeneral [user] (B)
  endproc
```

Defines:
`UserGeneral`, never used.
Uses `BaseId` 23a, `Proposal` 25b, and `Request` 27.

72 ⟨*AllUserGeneral process* 72⟩≡

```
  process AllUserGeneral [user] : noexit :=
  let p1 : Proposal = assert(black) + <>,
      p2 : Proposal = assert(white) + <>
  in
    user ? b1 : BaseId !doevaluate !p1; AllUserGeneral [user]
    []
    user ? b1 : BaseId !doevaluate !p2; AllUserGeneral [user]
    []
    user ? b1 : BaseId !doachieve !p1; AllUserGeneral [user]
    []
    user ? b1 : BaseId !doachieve !p2; AllUserGeneral [user]
    []
    user ? b1 : BaseId !dodeny ?r1 : Request;  AllUserGeneral [user]
    []
    user ? b1 : BaseId !doaccept ?r1 : Request;  AllUserGeneral [user]
    []
    user ? b1 : BaseId !doreject ?r1 : Request;  AllUserGeneral [user]
    []
    user ? b1 : BaseId !dochallenge !reqachieve(p1) !p2; AllUserGeneral [user]
    []
```

```
    user ? b1 : BaseId !dochallenge !reqachieve(p2) !p1; AllUserGeneral [user]
    []
    user ? b1 : BaseId !doforward !reqachieve(p1); AllUserGeneral [user]
    []
    user ? b1 : BaseId !doforward !reqachieve(p2); AllUserGeneral [user]
  endproc
```

Defines:
`AllUserGeneral`, used in chunks 80a and 82c.
Uses `BaseId` 23a, `Proposal` 25b, and `Request` 27.

User Consistency Tests `UserConsistency` only does *achieve* requests with two different and incompatible proposals. It is intended to search for violations of knowledge consistency.

73 ⟨*UserConsistency process* 73⟩≡

```
  process UserConsistency [user] (B : BaseId) : noexit :=
  user !B !doachieve !assert(white) + <>; UserConsistency [user] (B)
  []
  user !B !doachieve !assert(black) + <>; UserConsistency [user] (B)
  endproc
```

Defines:
`UserConsistency`, never used.
Uses `BaseId` 23a.

A.6.2 Constraints on parallelism

The following process is used to further restrict the possible behaviours of the system. It deliberately eliminates some perfectly valid behaviours; it must be asserted in some way that the reduced behaviour is still representative.

Serialize individual bases The process `SerializeIndividualKB` strongly reduces the possible interleavings between the behaviours of concurrent individual bases. It forces a strictly sequential application of the rules, relying on the fact that individual KB rules are either:

1. a user request followed by sending of a message to the group base,
2. reception of a message from the group base.

The soundness of this restriction is supported by the following arguments:

1. Individual bases are fully interleaved; user interactions are independent events.
2. Upward messages are always accepted by the upward queue.

74a ⟨*serialize process* 74a⟩≡

```
process SerializeIndividualKB [user,parentin,parentout] : exit :=
  exit
  []
  user ?b: BaseId ?act: UserAction ?b1: BaseId;
  parentout !b ?b2: BaseId ?msg: Message;
  SerializeIndividualKB [user,parentin,parentout]
  []
  user ?b: BaseId ?act: UserAction ?p1: Proposal;
  parentout !b ?b2: BaseId ?msg: Message;
  SerializeIndividualKB [user,parentin,parentout]
  []
  user ?b: BaseId ?act: UserAction ?r1 : Request;
  parentout !b ?b2: BaseId ?msg: Message;
  SerializeIndividualKB [user,parentin,parentout]
  []
  user ?b: BaseId ?act: UserAction ?r1: Request ?p1 : Proposal;
  parentout !b ?b2: BaseId ?msg: Message;
  SerializeIndividualKB [user,parentin,parentout]
  []
  parentin ?b1: BaseId ?b2: BaseId ?msg: Message;
  SerializeIndividualKB [user,parentin,parentout]
endproc
```

Defines:
`SerializeIndividualKB`, used in chunks 77b, 78a, 80b, and 83b.
Uses `BaseId` 23a, `Message` 28b, `Proposal` 25b, `Request` 27, and `UserAction` 30a.

A.6.3 System Scenarios

This section defines hierarchies of bases of different sizes and structures, then the user input scenarios that can be applied to them.

Hierarchy 1: 1I+1G The simplest non-trivial situation: one individual base, one group base. Already allows interesting observations.

$$\underbrace{base1}_{base3}$$

74b ⟨*KBHierarchy process 1* 74b⟩≡

```
process KBHierarchy [user,up1,down1,up2,down2,signal] : exit :=
  InitIndividualKB [user,down1,up1,signal] (base1)
  |[up1,down1]|
  ( hide up in InitQueue [up1,up] |[up]|
```

```
      InitGroupKB [up,down1,down2,up2,signal] (base3) )
    |[up2,down2]|
    exit
  endproc
```

Uses `InitGroupKB` 38, `InitIndividualKB` 37, and `InitQueue` 42c.

75a ⟨*KB gates 1* 75a⟩≡

```
  up1,down1,up2,down2
```

Scenario 1.1: 1I+1G, one achieve Register then achieve one proposal. Suitable for exhaustive generation.

75b ⟨*UserInput process 1.1* 75b⟩≡

```
  process UserInput [user] : noexit :=

      user !base1 !doregister !base3;
      user !base1 !doachieve !assert(square) + <>;
      stop
      |||
      AllUserReply [user]

  where

    ⟨AllUserReply process 70b⟩

  endproc
```

Uses `AllUserReply` 70b.

Scenario 1.2: 1I+1G, contradictory proposals Register then achieve two contradictory proposals. Used to study violations of knowledge consistency. Suitable for exhaustive generation.

75c ⟨*UserInput process 1.2* 75c⟩≡

```
  process UserInput [user] : noexit :=

      user !base1 !doregister !base3;
      user !base1 !doachieve !assert(white) + <>;
      user !base1 !doachieve !assert(black) + <>;
      stop
      |||
      AllUserReply [user]
```

```
  where

    ⟨AllUserReply process 70b⟩

  endproc
```

Uses `AllUserReply` 70b.

Scenario 1.3: 1I+1G, three achieves Register then achieve three proposals. Suitable for exhaustive generation; intended to demonstrate exponential blow-up.

76a ⟨*UserInput process 1.3* 76a⟩≡

```
  process UserInput [user] : noexit :=

      user !base1 !doregister !base3;
      user !base1 !doachieve !assert(white) + <>;
      user !base1 !doachieve !assert(black) + <>;
      user !base1 !doachieve !assert(square) + <>;
      stop
      |||
      AllUserReply [user]

  where

    ⟨AllUserReply process 70b⟩

  endproc
```

Uses `AllUserReply` 70b.

Hierarchy 1a: 1I+1G, no queue Same as hierarchy 1 but without queues, as in the first version of this specification. The resulting model contains numerous deadlocks due to simultaneous output attempts.

76b ⟨*KBHierarchy process 1a* 76b⟩≡

```
  process KBHierarchy [user,up1,down1,up2,down2,signal] : exit :=
    InitIndividualKB [user,down1,up1,signal] (base1)
    |[up1,down1]|
    InitGroupKB [up1,down1,down2,up2,signal] (base3)
    |[up2,down2]|
    exit
  endproc
```

Uses `InitGroupKB` 38 and `InitIndividualKB` 37.

77a ⟨*KB gates 1a* 77a⟩≡

```
up1,down1,up2,down2
```

Hierarchy 2: 2I+1G Same as previous but with two individual bases. Uses `SerializeIdividualKB` to reduce state space explosion.

$$\underbrace{base1 \quad base2}_{base3}$$

77b ⟨*KBHierarchy process 2* 77b⟩≡

```
process KBHierarchy [user,up1,down1,up2,down2,signal] : exit :=
  (
    (
      InitIndividualKB [user,down1,up1,signal] (base1)
      |||
      InitIndividualKB [user,down1,up1,signal] (base2)
    )
    |[user,down1,up1]|
    SerializeIndividualKB [user,down1,up1]
  )
  |[up1,down1]|
  ( hide up in InitQueue [up1,up] |[up]|
    InitGroupKB [up,down1,down2,up2,signal] (base3) )
  |[up2,down2]|
  exit (* group has no parent *)
where

  ⟨serialize process 74a⟩

endproc
```

Uses `InitGroupKB` 38, `InitIndividualKB` 37, `InitQueue` 42c, and `SerializeIndividualKB` 74a.

77c ⟨*KB gates 2* 77c⟩≡

```
up1,down1,up2,down2
```

Scenario 2.1: 2I+1G, one achieve Register then achieve one proposal. Suitable for exhaustive generation.

77d ⟨*UserInput process 2.1* 77d⟩≡

```
process UserInput [user] : noexit :=

    user !base1 !doregister !base3;
```

```
    user !base2 !doregister !base3;
    user !base1 !doachieve !assert(square) + <>;
    stop
    |||
    AllUserReply [user]

where

  ⟨AllUserReply process 70b⟩

endproc
```

Uses `AllUserReply` 70b.

Hierarchy 2a: 2I+1G, downward queues Same as hierarchy 2 but using a queue for downward iso. upward traffic. Experience shows that this produces larger transition systems than the previous one. To be used with the same user environments as hierarchy 2.

78a ⟨*KBHierarchy process 2a* 78a⟩≡

```
process KBHierarchy [user,up1,down1,up2,down2,signal] : exit :=
  (
    (
      InitIndividualKB [user,down1,up1,signal] (base1)
      |||
      InitIndividualKB [user,down1,up1,signal] (base2)
    )
    |[user,down1,up1]|
    SerializeIndividualKB [user,down1,up1]
  )
  |[up1,down1]|
  ( hide down in InitQueue [down,down1] |[down]|
    InitGroupKB [up1,down,down2,up2,signal] (base3) )
  |[up2,down2]|
  exit (* group has no parent *)
where

  ⟨serialize process 74a⟩

endproc
```

Uses `InitGroupKB` 38, `InitIndividualKB` 37, `InitQueue` 42c, and `SerializeIndividualKB` 74a.

78b ⟨*KB gates 2a* 78b⟩≡

```
up1,down1,up2,down2
```

Hierarchy 3: 1I+2G Simplest three-level hierarchy with one individual and two group bases:

$$\underbrace{\underbrace{base1}_{base3}}_{base5}$$

79a ⟨*KBHierarchy process 3* 79a⟩≡

```
process KBHierarchy [user,up1,down1,up2,down2,up3,down3,signal] : exit :=
  InitIndividualKB [user,down1,up1,signal] (base1)
  |[up1,down1]|
  ( hide up in InitQueue [up1,up] |[up]|
    InitGroupKB [up,down1,down2,up2,signal] (base3) )
  |[up2,down2]|
  ( hide up in InitQueue [up2,up] |[up]|
    InitGroupKB [up,down2,down3,up3,signal] (base5) )
  |[down3,up3]|
  exit (* base5 has no parent *)
where

  ⟨serialize process 74a⟩

endproc
```

Uses `InitGroupKB` 38, `InitIndividualKB` 37, and `InitQueue` 42c.

79b ⟨*KB gates 3* 79b⟩≡

```
up1,down1,up2,down2,up3,down3
```

Scenario 3.1: 1I+2G, one achieve Register then achieve one proposal in `base3`.

79c ⟨*UserInput process 3.1* 79c⟩≡

```
process UserInput [user] : noexit :=

    user !base1 !doregister !base3;
    user !base1 !doforward !reqregister(base5);
    user !base1 !doforward !reqachieve(assert(white) + <>);
    stop
    |||
    AllUserReply [user]

where

  ⟨AllUserReply process 70b⟩
```

```
  endproc
```

Uses `AllUserReply` 70b.

Scenario 3.2: 1I+2G, general user Uses `UserGeneral` and therefore usable for interactive simulation or strongly confined exploration only.

80a ⟨*UserInput process 3.2* 80a⟩≡

```
  process UserInput [user] : noexit :=

      user !base1 !doregister !base3;
      user !base1 !doforward !reqregister(base5);
      stop
      |||
      AllUserGeneral [user]

  where

    ⟨AllUserReply process 70b⟩

  endproc
```

Uses `AllUserGeneral` 72.

Hierarchy 4: 3I+2G Three-level hierarchy with three individual and two group bases:

$$\underbrace{\underbrace{base1 \quad base2}_{base3} \quad base4}_{base5}$$

Still uses `SerializeIndividualKB`, but only on `base1` and `base2` — there is no simple way to serialize `base4` w.r.t. its peer subgroup.

80b ⟨*KBHierarchy process 4* 80b⟩≡

```
  process KBHierarchy [user,up1,down1,up2,down2,up3,down3,signal] : exit :=
    (
      (
        (
          (
            InitIndividualKB [user,down1,up1,signal] (base1)
            |||
            InitIndividualKB [user,down1,up1,signal] (base2)
```

```
        )
        |[user,down1,up1]|
        SerializeIndividualKB [user,down1,up1]
      )
      |[up1,down1]|
      ( hide up in InitQueue [up1,up] |[up]|
        InitGroupKB [up,down1,down2,up2,signal] (base3) )
    )
    |||
    InitIndividualKB [user,down2,up2,signal] (base4)
  )
  |[up2,down2]|
  ( hide up in InitQueue [up2,up] |[up]|
    InitGroupKB [up,down2,down3,up3,signal] (base5) )
  |[down3,up3]|
  exit (* base5 has no parent *)
where

  ⟨serialize process 74a⟩

endproc
```

Uses `InitGroupKB` 38, `InitIndividualKB` 37, `InitQueue` 42c, and `SerializeIndividualKB` 74a.

81a ⟨*KB gates 4* 81a⟩≡

```
up1,down1,up2,down2,up3,down3
```

Hierarchy 4a: 3I+2G, full interleave Same as hierarchy 4 but without `SerializeIndividualKB`.

81b ⟨*KBHierarchy process 4a* 81b⟩≡

```
process KBHierarchy [user,up1,down1,up2,down2,up3,down3,signal] : exit :=
  (
    (
      (
        InitIndividualKB [user,down1,up1,signal] (base1)
        |||
        InitIndividualKB [user,down1,up1,signal] (base2)
      )
      |[up1,down1]|
      ( hide up in InitQueue [up1,up] |[up]|
        InitGroupKB [up,down1,down2,up2,signal] (base3) )
    )
    |||
    InitIndividualKB [user,down2,up2,signal] (base4)
```

```
    )
    |[up2,down2]|
    ( hide up in InitQueue [up2,up] |[up]|
      InitGroupKB [up,down2,down3,up3,signal] (base5) )
    |[down3,up3]|
    exit (* base5 has no parent *)

  endproc
```

Uses `InitGroupKB` 38, `InitIndividualKB` 37, and `InitQueue` 42c.

82a ⟨*KB gates 4a* 82a⟩≡

```
  up1,down1,up2,down2,up3,down3
```

Scenario 4.1: 3I+2G, one achieve Register then achieve one proposal in `base5`. Too large for exhaustive generation, at time of writing.

82b ⟨*UserInput process 4.1* 82b⟩≡

```
  process UserInput [user] : noexit :=

      user !base1 !doregister !base3;
      user !base2 !doregister !base3;
      user !base4 !doregister !base5;
      user !base1 !doforward !reqregister(base5);
      user !base1 !doforward !reqachieve(assert(white) + <>);
      stop
      |||
      AllUserReply [user]

  where

    ⟨AllUserReply process 70b⟩

  endproc
```

Uses `AllUserReply` 70b.

Scenario 4.2: 3I+2G, general user Uses `UserGeneral` and therefore usable for interactive simulation or strongly confined exploration only.

82c ⟨*UserInput process 4.2* 82c⟩≡

```
  process UserInput [user] : noexit :=

      user !base1 !doregister !base3;
```

```
      user !base1 !doforward !reqregister(base5);
      stop
      |||
      user !base2 !doregister !base3;
      stop
      |||
      user !base4 !doregister !base5;
      stop
      |||
      AllUserGeneral [user]

  where

    ⟨AllUserGeneral process 72⟩

  endproc
```

Uses `AllUserGeneral` 72.

Scenario 4.3: 3I+2G, one achieve with all accept The minimal interesting case. There is a hope to generate the model for this one, though this could not be achieved at time of writing.

83a ⟨*UserInput process 4.3* 83a⟩≡

```
  process UserInput [user] : noexit :=

      user !base1 !doregister !base3;
      user !base2 !doregister !base3;
      user !base4 !doregister !base5;
      user !base1 !doforward !reqregister(base5);
      stop
      |||
      AllUserAccept [user]

  where

    ⟨AllUserAccept process 71b⟩

  endproc
```

Uses `AllUserAccept` 71b.

Hierarchy 5: 3I+2G, Registered Same as hierarchy 4 but with already registered bases, to prune the registering phase of the scenario.

83b ⟨*KBHierarchy process 5* 83b⟩≡

```
process KBHierarchy [user,up1,down1,up2,down2,up3,down3,signal] : exit :=
  (
    (
      (
        (
          IdleIndividualKB [user,down1,up1,signal] (base1, some(base3))
          |||
          IdleIndividualKB [user,down1,up1,signal] (base2, some(base3))
        )
        |[user,down1,up1]|
        SerializeIndividualKB [user,down1,up1]
      )
      |[up1,down1]|
      ( hide up in InitQueue [up1,up] |[up]|
        IdleGroupKB [up,down1,down2,up2,signal] (base3, some(base5),
          insert(base1, insert(base2, {}))) )
    )
    |||
    IdleIndividualKB [user,down2,up2,signal] (base4, some(base5))
  )
  |[up2,down2]|
  ( hide up in InitQueue [up2,up] |[up]|
    IdleGroupKB [up,down2,down3,up3,signal] (base5, none,
      insert(base3, insert(base4, {}))) )
  |[down3,up3]|
  exit (* base5 has no parent *)
where

  ⟨serialize process 74a⟩

endproc
```

Uses `InitQueue` 42c and `SerializeIndividualKB` 74a.

84a ⟨*KB gates 5* 84a⟩≡

```
up1,down1,up2,down2,up3,down3
```

Scenario 5.1: 3I+2G, one achieve Achieve one proposal in `base5`. Exhaustive generation is achievable (albeit lengthy).

84b ⟨*UserInput process 5.1* 84b⟩≡

```
process UserInput [user] : noexit :=
```

```
    user !base1 !doforward !reqachieve(assert(white) + <>);
    stop
    |||
    AllUserReply [user]

where

  ⟨AllUserReply process 70b⟩

endproc
```

Uses `AllUserReply` 70b.

Scenario 5.2: 3I+2G, one achieve, separate users Same as scenario 5.1 but uses separate `UserReply` processes for each user instead of a single `AllUserReply` process. For performance comparisons.

85 ⟨*UserInput process 5.2* 85⟩≡

```
process UserInput [user] : noexit :=

    user !base1 !doforward !reqachieve(assert(white) + <>);
    stop
    |||
    UserReply [user] (base1)
    |||
    UserReply [user] (base2)
    |||
    UserReply [user] (base4)

where

  ⟨UserReply process 70a⟩

endproc
```

Uses `UserReply` 70a.

Hierarchy 6: 1I+3G, competing groups A three-level hierarchy with two competing group bases at root level. Intended to study concurrent $forward(register)$ requests.

$$\underbrace{\underbrace{base1}_{base3}}_{\overbrace{base5 \quad base6}}$$

86a ⟨*KBHierarchy process 6* 86a⟩≡

```
process KBHierarchy [user,up1,down1,up2,down2,up3,down3,signal] : exit :=
  InitIndividualKB [user,down1,up1,signal] (base1)
  |[up1,down1]|
  ( hide up in InitQueue [up1,up] |[up]|
    InitGroupKB [up,down1,down2,up2,signal] (base3) )
  |[up2,down2]|
  ( hide up in InitQueue [up2,up] |[up]|
    (
      InitGroupKB [up,down2,down3,up3,signal] (base5)
      |||
      InitGroupKB [up,down2,down3,up3,signal] (base6)
    )
  )
  |[down3,up3]|
  exit (* no parent *)
where

  ⟨serialize process 74a⟩

endproc
```

Uses `InitGroupKB` 38, `InitIndividualKB` 37, and `InitQueue` 42c.

86b ⟨*KB gates 6* 86b⟩≡

```
up1,down1,up2,down2,up3,down3
```

Scenario 6.1: 1I+3G, two forward(achieve) with all accept User tries to subscribe the intermediate group base to two groups simultaneously.

86c ⟨*UserInput process 6.1* 86c⟩≡

```
process UserInput [user] : noexit :=

    user !base1 !doregister !base3;
    user !base1 !doforward !reqregister(base5);
    user !base1 !doforward !reqregister(base6);
    stop
    |||
    AllUserAccept [user]

where

  ⟨AllUserAccept process 71b⟩
```

```
  endproc
```

Uses `AllUserAccept` 71b.

A.7 Index of Lotos Definitions

B Detailed Verification Results

This appendix provides a representative sample of the results obtained from the LOTOS specification, in the form of diagnostic traces produced by EXHIBITOR. Each sub-section has the following structure:

Specification Variant: Hierarchy x, Scenario y.
The variant of the specification from which the results are obtained. The numbers x and y refer to the definitions in Appendix A.

Trace Pattern:

```
...
```

The pattern used to obtain the traces, in EXHIBITOR's input format.

Results:

```
*** sequence found at depth ...

...
```

The traces matching the given pattern, as reported by EXHIBITOR.

B.1 Dialogue of the Deaf

Specification Variant: Hierarchy 1a, Scenario 1.1.

One individual and one group base, directly synchronized (no buffering). User registers then submits a single proposal.

Trace Pattern:

```
<while> <any>
<deadlock>
```

Any sequence leading to deadlock.

Results:

```
*** sequence found at depth 8

<initial state>
"USER !1 !DOREGISTER !3"
"UP1 !1 !3 !REGISTER (0, 3)"
"DOWN1 !3 !1 !NOTIFY (0, ACCEPTX)"
"DOWN1 !3 !1 !TELL (<>)"
"USER !1 !DOACHIEVE !+ (ASSERT (SQUARE), <>)"
"SIGNAL !3 !SIGREGISTERED !1"
"UP1 !1 !3 !ACHIEVE (1, + (ASSERT (SQUARE), <>))"
"USER !1 !DODENY !REQACHIEVE (+ (ASSERT (SQUARE), <>))"
<deadlock>
```

A deadlock due to the lack of asynchronism: both bases 1 and 3 are willing to send a message to each other while not willing to receive. The end of the trace is illustrated on Figure 5. Five other traces were found (one at depth 4, four at depth 12).

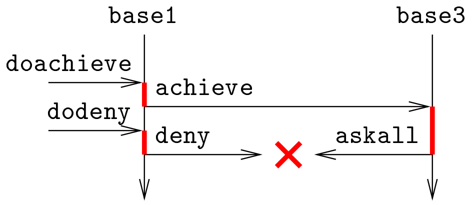

Figure 5: "Dialogue of the Deaf" deadlock

B.2 Unexpected Receptions

Specification Variant: Hierarchy 3, Scenario 3.1.

A three-level hierarchy with one individual and two group bases (resp. `1`, `3` and `5`). User registers `1` to `3` and `3` to `5`, then submits a single proposal to `5`.

Trace Pattern:

```
<until> [.*SIGNOTINA.*ERROR.*]
```

Find a `signotina` signal for an `error` message. `signotina` reports messages that cannot be dealt with because the corresponding request cannot be found in table A.

Results:

```
*** sequence found at depth 14

<initial state>
"USER !1 !DOREGISTER !3"
"UP1 !1 !3 !REGISTER (0, 3)"
"DOWN1 !3 !1 !NOTIFY (0, ACCEPTX)"
"DOWN1 !3 !1 !TELL (<>)"
"USER !1 !DOFORWARD !REQREGISTER (5)"
"SIGNAL !3 !SIGREGISTERED !1"
"UP1 !1 !3 !FORWARD (1, REQREGISTER (5))"
"USER !1 !DOFORWARD !REQACHIEVE (+ (ASSERT (WHITE), <>))"
"UP1 !1 !3 !FORWARD (2, REQACHIEVE (+ (ASSERT (WHITE), <>)))"
"USER !1 !DODENY !REQFORWARD (REQACHIEVE (+ (ASSERT (WHITE), <>)))"
"UP1 !1 !3 !DENY (2)"
"DOWN1 !3 !1 !ASKALL (0, REQFORWARD (REQREGISTER (5)))"
"DOWN1 !3 !1 !ERROR (2)"
"SIGNAL !1 !SIGNOTINA !ERROR (2)"
<goal state>
```

The proposal to base `5` produces an `error` message from `3`, because registration of `3` to `5` is not fulfilled yet. Meanwhile, `1` withdraws its proposal with a `deny` message and then wipes it out of its local A table. As shown on Figure 6, the `deny` and `error` messages cross each other so that `1` has lost trace of its request when it receives the error message.

Three further cases, not listed here, were found using the following patterns:

```
<until> [.*SIGNOTINC.*REPLY.*]
[]
<until> [.*SIGNOTINA.*NOTIFY.*]
[]
<until> [.*SIGNOTINC.*DENY.*]
```

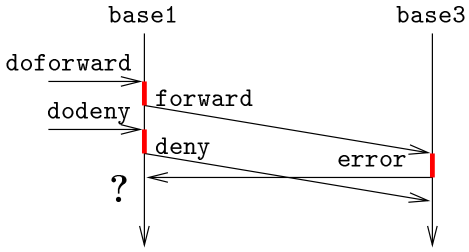

Figure 6: Unexpected reception

Finally, the following pattern (re-formatted for readability) searches for other cases than the four previous ones. No matching trace could be found in any of the tested scenarios.

```
<until> [SIGNAL.*SIGNOTIN.*]
        & ~[.*SIGNOTINC.*REPLY.*]
        & ~[.*SIGNOTINA.*ERROR.*]
        & ~[.*SIGNOTINC.*DENY.*]
        & ~[.*SIGNOTINA.*NOTIFY.*]
```

B.3 Knowledge Inconsistency

Specification Variant: Hierarchy 1, Scenario 1.2.

One individual and one group base. User registers then submits two proposals asserting contradictory atoms `assert(white)` and `assert(black)`.

Trace Pattern:

```
<until> [SIGNAL.*SIGSTORED.*FALSE]
```

Find a `sigstored` signal with attribute `false`, indicating an inconsistent update of a knowledge repository.

Results:

```
*** sequence found at depth 23

<initial state>
"USER !1 !DOREGISTER !3"
"UP1 !1 !3 !REGISTER (0, 3)"
"DOWN1 !3 !1 !NOTIFY (0, ACCEPTX)"
"DOWN1 !3 !1 !TELL (<>)"
"USER !1 !DOACHIEVE !+ (ASSERT (WHITE), <>)"
"SIGNAL !3 !SIGREGISTERED !1"
"UP1 !1 !3 !ACHIEVE (1, + (ASSERT (WHITE), <>))"
"USER !1 !DOACHIEVE !+ (ASSERT (BLACK), <>)"
"UP1 !1 !3 !ACHIEVE (2, + (ASSERT (BLACK), <>))"
"DOWN1 !3 !1 !ASKALL (0, REQACHIEVE (+ (ASSERT (WHITE), <>)))"
"USER !1 !DOACCEPT !REQACHIEVE (+ (ASSERT (WHITE), <>))"
"UP1 !1 !3 !REPLY (0, ACCEPTX)"
"DOWN1 !3 !1 !ASKALL (1, REQACHIEVE (+ (ASSERT (BLACK), <>)))"
"DOWN1 !3 !1 !NOTIFY (1, ACCEPTX)"
"DOWN1 !3 !1 !POOLNOTIFY (0, ACCEPTX)"
"USER !1 !DOACCEPT !REQACHIEVE (+ (ASSERT (BLACK), <>))"
"UP1 !1 !3 !REPLY (1, ACCEPTX)"
"DOWN1 !3 !1 !TELL (+ (ASSERT (WHITE), <>))"
"SIGNAL !3 !SIGSTORED !+ (ASSERT (WHITE), <>) !TRUE"
"DOWN1 !3 !1 !NOTIFY (2, ACCEPTX)"
"DOWN1 !3 !1 !POOLNOTIFY (1, ACCEPTX)"
"DOWN1 !3 !1 !TELL (+ (ASSERT (BLACK), <>))"
"SIGNAL !3 !SIGSTORED !+ (ASSERT (BLACK), <>) !FALSE"
<goal state>
```

This trace is summarized in Figure 7. Essentially, both proposals are voted concurrently, and since base 3 checks the consistency w.r.t. its own repository *before* starting the vote, the contradiction is not detected and both end up being added to the repository.

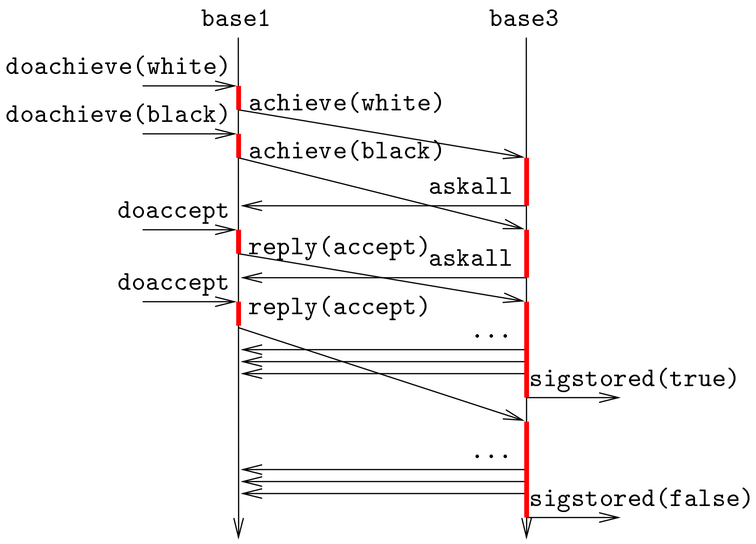

Figure 7: Inconsistent knowledge

B.4 Hierarchy Inconsistency

Specification Variant: Hierarchy 6, Scenario 6.1.

A three-level hierarchy with an individual base 1, an intermediate group base 3 and two competing root group bases 5 and 6. The user registers 1 to 3 and then attempts to register 3 to both 5 and 6.

Trace Pattern:

```
<until> [.*SIGREGISTERED !3]
<until> [.*SIGREGISTERED !3]
<until> "OK"
```

Find a successfully terminating trace where 3 registers twice.

Results:

```
*** sequence found at depth 40

<initial state>
"USER !1 !DOREGISTER !3"
"UP1 !1 !3 !REGISTER (0, 3)"
"DOWN1 !3 !1 !NOTIFY (0, ACCEPTX)"
"DOWN1 !3 !1 !TELL (<>)"
"USER !1 !DOFORWARD !REQREGISTER (5)"
"SIGNAL !3 !SIGREGISTERED !1"
"UP1 !1 !3 !FORWARD (1, REQREGISTER (5))"
"USER !1 !DOFORWARD !REQREGISTER (6)"
"UP1 !1 !3 !FORWARD (2, REQREGISTER (6))"
"DOWN1 !3 !1 !ASKALL (0, REQFORWARD (REQREGISTER (5)))"
"USER !1 !DOACCEPT !REQFORWARD (REQREGISTER (5))"
"UP1 !1 !3 !REPLY (0, ACCEPTX)"
"DOWN1 !3 !1 !ASKALL (1, REQFORWARD (REQREGISTER (6)))"
"DOWN1 !3 !1 !POOLNOTIFY (0, ACCEPTX)"
"UP2 !3 !5 !REGISTER (0, 5)"
"DOWN2 !5 !3 !NOTIFY (0, ACCEPTX)"
"DOWN1 !3 !1 !NOTIFY (1, ACCEPTX)"
"DOWN2 !5 !3 !TELL (<>)"
"SIGNAL !5 !SIGREGISTERED !3"
"DOWN1 !3 !1 !ASKALL (2, REQTELL (<>))"
"USER !1 !DOACCEPT !REQTELL (<>)"
"UP1 !1 !3 !REPLY (2, ACCEPTX)"
"USER !1 !DOACCEPT !REQFORWARD (REQREGISTER (6))"
"UP1 !1 !3 !REPLY (1, ACCEPTX)"
```

```
"DOWN1 !3 !1 !POOLNOTIFY (2, ACCEPTX)"
"DOWN1 !3 !1 !TELL (<>)"
"SIGNAL !3 !SIGSTORED !<> !TRUE"
"DOWN1 !3 !1 !POOLNOTIFY (1, ACCEPTX)"
"UP2 !3 !6 !REGISTER (1, 6)"
"DOWN2 !6 !3 !NOTIFY (1, ACCEPTX)"
"DOWN1 !3 !1 !NOTIFY (2, ACCEPTX)"
"DOWN2 !6 !3 !TELL (<>)"
"SIGNAL !6 !SIGREGISTERED !3"
"DOWN1 !3 !1 !ASKALL (3, REQTELL (<>))"
"USER !1 !DOACCEPT !REQTELL (<>)"
"UP1 !1 !3 !REPLY (3, ACCEPTX)"
"DOWN1 !3 !1 !POOLNOTIFY (3, ACCEPTX)"
"DOWN1 !3 !1 !TELL (<>)"
"SIGNAL !3 !SIGSTORED !<> !TRUE"
"OK"
<goal state>
```

This long trace is summarized in Figure 8. Again, base 3 checks that it is not already registered *before* starting the vote on the registration requests, so both votes can proceed to completion concurrently.

Other executions of the same scenario leading to deadlocks have also been found. The shortest one is at depth 26 and results from the same kind of situation.

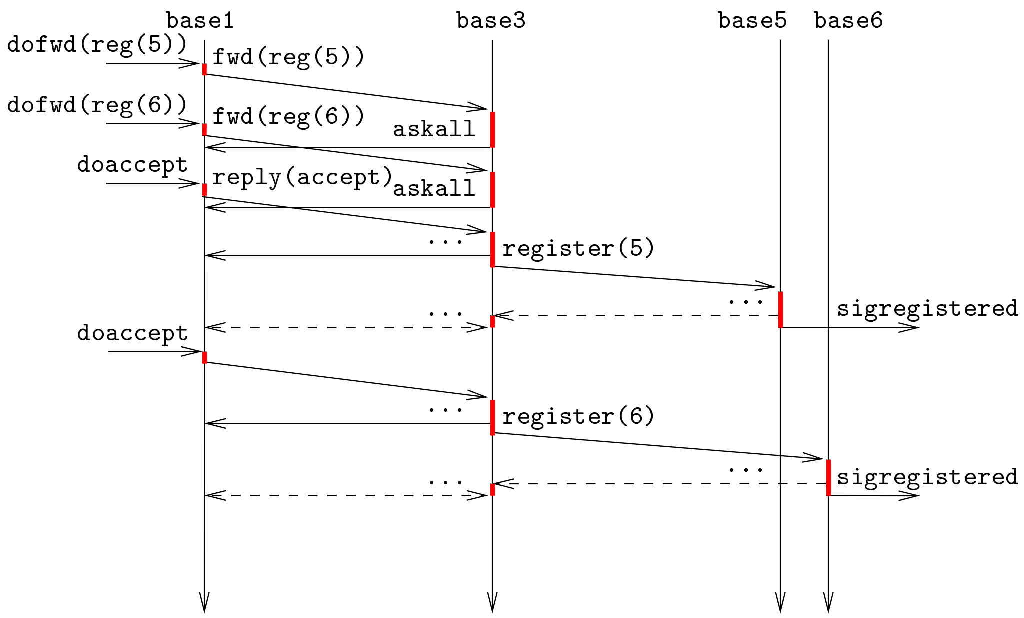

Figure 8: Double registration

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