Formal Reasoning about Communication Systems II

Automated Fast-Track Reconfiguration

Christoph Kreitz

Abstract. We present formal techniques for improving the performance of group communication systems built with the ENSEMBLE toolkit. For common sequences of operations we identify a fast-track through a stack of communication protocols and reconfigure the system's code accordingly. Our techniques are implemented as fully automated tactics of the NUPRL proof development system and are based on an embedding the implementation language of ENSEMBLE into the logical language of NUPRL. Together with verification techniques to be developed in the near future they will lead to a logical programming environment for the construction of reliable and efficient group communication systems.

Technical Report CORNELL CS:TR98-17xx
July, 1997

Department of Computer Science
Cornell University
Ithaca, New York

Contents

List of Figures

Formal Reasoning about Communication Systems II

Automated Fast-Track Reconfiguration

Christoph Kreitz

Department of Computer Science
Cornell-University
Ithaca, NY 14853-7501
kreitz@cs.cornell.edu

Abstract. We present formal techniques for improving the performance of group communication systems built with the ENSEMBLE toolkit. For common sequences of operations we identify a fast-track through a stack of communication protocols and reconfigure the system's code accordingly. Our techniques are implemented as fully automated tactics of the NUPRL proof development system and are based on an embedding the implementation language of ENSEMBLE into the logical language of NUPRL. Together with verification techniques to be developed in the near future they will lead to a logical programming environment for the construction of reliable and efficient group communication systems.

1 Introduction

Due to the wide range of safety-critical applications of group communication [Bir97], the development of a secure and reliable communications infrastructure has become increasingly important. Current networks, however, are inadequate for this purpose because considerable technical challenges could not be overcome yet. First, there is the *secure implementation problem*: correctly implementing distributed systems is notoriously difficult [Bir97,CHTC96]. Although it is possible to prove the correctness of theoretical algorithms [Rus92,LR93,AH97,Rus97] it is difficult to transform these idealizations into implementations of real systems. The *formalization barrier*, on the other hand, prevents formal tools for checking software correctness from being used to maximum benefit. These tools are computationally costly, difficult to understand, and not yet integrated into software development environments. Finally, there is the *performance cost* of modularity. To maximize clarity and code re-use, systems are divided into clean modules, which can operate in a broad number of environments. But when modules are combined in a restricted context, much of the code becomes useless or redundant and leads to unnecessary large execution times.

To overcome these problems, we have linked ENSEMBLE [Hay96,Hay98b], a flexible group communication toolkit, to NUPRL [CAB+86], a proof system for mathematical reasoning about programs. By establishing this this link [Kre97] we have created a formal reasoning environment that can deal with the actual ENSEMBLE code, which has substantially lowered the formalization barrier. Within the reasoning environment we can then address the secure implementation problem by developing tactics for a formal verification of critical system properties in NUPRL. Furthermore, we are able to reduce the performance cost of group communication systems by applying reconfiguration strategies to the system's code and exporting their results back into the OCAML environment.

Ensemble is the third generation of group communication systems developed at Cornell that aim at securing critical networked applications. The first system, ISIS [BvR94], became one of the first widely adopted technologies in this area and found its way into the Swiss and New York Stock Exchanges, the French Air Traffic Control System, and other safety-critical applications. The HORUS system [vRBM96], a modular redesign of ISIS, is based on layering communication protocols, which can be combined almost arbitrarily to match the needs of a particular application. The protocol stacking architecture makes HORUS more flexible and also faster than ISIS, as the efficiency of its protocol stacks can be improved by analyzing common sequences of operation and *reconfiguring* the system code accordingly [HvR96].

Reconfiguring HORUS protocol stacks, however, is difficult and error prone, as the layer code is written in C and too complex to reason about. Concerns about the reliability of such a technology base for secure networked applications led to the implementation of ENSEMBLE [Hay96,Hay98a,Hay98b], which is based on HORUS but coded almost entirely in the high-level programming language OCAML [Ler97], a member of the ML [GMW79] language family. Due to the use of ML, ENSEMBLE turned out to be one of the most scalable and portable, but also one of the fastest existing reliable multicast systems. But the main reason for choosing OCAML was to enable formal reasoning about ENSEMBLE's code within a theorem proving environment and to use methods from automated deduction for reconfiguring the system and verifying its properties [Hay97,Hay98b].

Conceptually, each protocol stack is a finite I/O-automaton that could be handled by propositional methods. But reasoning about the *actual* ENSEMBLE *code* requires a proof environment capable of expressing the semantics of a real programming language and establishing a correspondence between the implementation of a protocol stack and its formal model.

The NuPRL proof development system [CAB$^+$86] is a framework for mathematical reasoning about programs and secure program transformations. Proof strategies, or *tactics*, can be tailored to follow the particular style of reasoning in distributed systems. Tactics also produce (possibly partial) proof objects, which can be used as documentation or reveal valuable debugging information if a verification did not succeed. Code optimizations can be achieved by applying *rewrite tactics*, which reconfigure the protocol stack of a given application.

Because of its expressive formal calculus, NUPRL is well suited for building a reasoning environment for ENSEMBLE. NUPRL's *Type Theory* already includes formalizations of the fundamental concepts of mathematics, programming, and data types. It also contains a functional programming language that corresponds to the core of ML. The NuPRL system supports interactive and semi-automatic formal reasoning, conservative language extensions by user-defined concepts, the evaluation of programs, and an extendable library of verified knowledge from various domains. These features make it possible to represent the code of Ensemble and its specifications as terms of NuPRL's formal language and to use NuPRL as a *logical programming environment* [KHH98] whose capabilities go beyond the usual type-checking and syntactical debugging capabilities of OCaml.

The formal link between Ensemble and NuPRL can be established because of the similarity between the programming language OCaml and NuPRL's type theory. To enable formal reasoning about Ensemble's implementation, we have to convert OCaml programs into terms of the logical language of NuPRL that capture the semantics of these programs. For this purpose we have developed a *type-theoretical semantics* of OCaml that is faithful with respect to the compiler and manual [Ler97]. Our formalization [Kre97, Section 3] is restricted to the subset of OCaml that is necessary for implementing finite state-event systems like Ensemble, i.e. the functional subset together with simple imperative features. Because of this limitation we are not forced to deal with aspects of the programming language that cause complications in a rigorous formalization and are not used when reasoning about protocol stacks.

We have 'implemented' this formalization using NuPRL's definition mechanism: each OCaml language construct is represented by a new NuPRL term that is defined to have the formal semantics of this construct. This *abstraction* is coupled with a *display form*, which makes sure that the formal representation has the same outer appearance as the original code. Thus a NuPRL term represents both the program text of an OCaml program and its formal semantics. Furthermore, the well-formedness and soundness of the new terms with respect to the rest of type theory is proved in separate theorem to make sure that such issues can be handled automatically during verifications and reconfigurations.

We have also developed a formal *programming logic* for OCaml by describing rules for reasoning about OCaml expressions and rules for symbolically evaluating them [Kre97, Section 4]. The rules were implemented as NuPRL tactics and are therefore correct with respect to the type-theoretical semantics of OCaml. As a result, formal reasoning about Ensemble's protocol stacks within NuPRL can be performed at the level of OCaml programs instead of type theory. All terms representing OCaml programs are displayed in OCaml syntax and individual reasoning and program transformation steps always preserve the "OCaml-ness" of the terms they are dealing with. This enables system experts who are not necessarily logicians to reason formally about OCaml-programs without having to understand the peculiarities of the proof environment.

Finally, we have created tools that convert OCaml programs into their formal NuPRL representations (and vice versa) and store them as objects of NuPRL's library [Kre97, Section 5]. Using these tools we are able to translate the complete OCaml-code of the Ensemble system into NuPRL and to keep track of modifications in Ensemble's implementation.

In this paper we will focus on formal techniques for improving the performance of Ensemble within our logical programming environment.

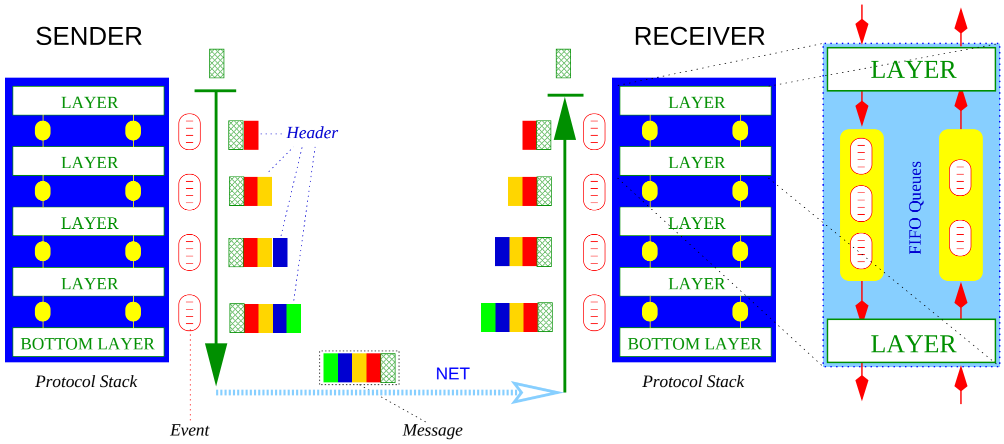

Fig. 1. Ensemble architecture: *Protocol layers are linked by FIFO event queues*

To ensure that Ensemble's protocol layers can be combined almost arbitrarily, each layer makes only very few assumptions about adjacent layers. As a result, all kinds of messages, including errors, must be handled within the layer. On the way down each layer adds a header to the message that indicates how the corresponding layer in the receiver's stack has to be activated (see Figure 1). Because of this protocol stacking architecture, Ensemble can easily be configured for many safety-critical network applications with many different properties. But the code of these application systems is not as efficient as a specialized implementation, as it contains redundancies and requires communication between layers. In the common case, however, the system is only used to send or broadcast a message that passes straight through the stack and affects only a few layers.

By *reconfiguring* a protocol stack [HvR96] one can achieve a more efficient treatment of the common cases of operation. For this purpose one has to analyze common sequences of operations, identify the structure of common messages, and characterize the normal status of a layer's state. Given these informations one can describe a *fast-track* through the protocol stack that can used instead of the full protocol stack whenever a message and the current state can be characterized as "common". Reconfiguring the code of the protocol stack accordingly can lead to dramatic efficiency improvements. Experiments [HvR96,Hay98b] have shown that a speedup of factor 30–50 can be achieved by function inlining, symbolic code evaluation, unfolding abstractions, elimination of redundancies and dead code, removing the communication overhead between layers, delaying state updates, and compressing the headers of standard messages before sending them over the net. These techniques, however, are not supported by current compilers and, due to the code size of typical applications, fast-track reconfigurations by hand are time consuming and contain a high risk of error. Without formal support, a reconfiguration of an application system would be infeasible.

We have developed a variety of formal techniques for the reconfiguration of networked application systems built with the Ensemble group communication toolkit. These techniques, which we will describe in this paper, are implemented as tactics of the NuPRL proof development system and can thus be included in the distribution of a logical programming environment for Ensemble.

In section 2 we will discuss the reconfiguration of individual protocol layers. As the code of the layers is known, the reconfigurations are done a priori with a small amount of interaction and can be included in the distribution. The reconfiguration of protocol stacks discussed in section 3, however, depends strongly on the application and must be fully automated. In section 4 we will present tactics for header compression that can be applied after a stack has been reconfigured. Section 5 describes how to create the Ocaml-code of an optimized protocol stack from the results of the logical reconfigurations. In section 6 we discuss the interface between the protocol stack and the application that uses it. The Ensemble code for building protocol stacks from individual layers as well as the formal definitions and theorems that would be included in a distribution of the logical programming environment are listed in the appendices A and B.

2 Reconfiguration of Protocol Layers

In principle, fast-track reconfigurations can be performed entirely by a few general evaluation techniques such as function inlining, symbolic evaluation through controlled β- and η-reduction, and knowledge-based simplification. These techniques can be formally based on the tactics for symbolically evaluating Ocaml-expressions mentioned in Section 1 and on conditional rewrite rules, which we implemented via substitution and lemma application.

We have developed a small set of fundamental evaluation tactics that automatically detect pieces of code that can be optimized and rewrite them accordingly. These tactics are very useful for the reconfiguration of individual protocol layers. But for the reconfiguration of protocol stacks containing thousands of lines of code they are not efficient enough since too much search is involved in the process. Therefore we have developed specialized reconfiguration tactics that follow the code structure of Ensemble's protocol layers and the function for layer composition and avoid search almost completely.

2.1 Tactic-based Fast-Track Reconfiguration

Figure 2 shows the typical structure of Ensemble's code for individual protocol layers. Apart from type declarations and a few auxiliary functions a layer `l` essentially consists of two functions `init` and `hdlrs`. The function `init` initializes the state of the layer according to a given view state `vs` and local state information `ls`. The `hdlrs` function describes how the layer's state is affected by an incoming event and which events will be sent to adjacent layers. For the latter, it describes how the event *handlers* of a protocol stack are transformed by inserting the layer `l` into the stack. For the sake of clarity the event handlers for up-going and down-going events are separated into five sub-handlers

```
let  name   = String containing the layer's name
type header = Variant record type for possible headers
type state  = Record type for the layer's state
let  init () (ls,vs) = {Initial state of the layer}
let  hdlrs s (ls,vs) {up_out=up;upnm_out=upnm;
                      dn_out=dn;dnlm_out=dnlm;dnnm_out=dnnm}
   = ...
     let up_hdlr ev abv hdr = ...
     and uplm_hdlr ev hdr   = ...
     and upnm_hdlr ev       = ...
     and dn_hdlr ev abv     = ...
     and dnnm_hdlr ev       = ...
     in {up_in=up_hdlr;uplm_in=uplm_hdlr;upnm_in=upnm_hdlr;
         dn_in=dn_hdlr;dnnm_in=dnnm_hdlr}
let l args vs = Layer.hdr init hdlrs args vs
Layer.install name l
```

Fig. 2. Code structure of Ensemble's protocol layers

dealing with regular events (`up,dn`), local events (`uplm,dnlm`), and events with empty headers (`upnm,dnnm`). The layer `l` itself is created from `init` and `hdlrs` through the function `Layer.hdr`, which glues these five sub-handlers together, and `installed` under its layer name. This technique makes it possible to `convert` the implementation of a layer into a functional (see Figure 3), imperative, or threaded version while keeping a single reference implementation of the layer itself. The main difference lies in the realization of event passing between layers. The reconfiguration tactics that we will describe in this section closely follow the structure of the layer code and of the functions `Layer.hdr` and `convert`.

In order to create a fast-path through a protocol layer we have to state the assumptions about common events and layer states and to optimize the event handler of the layer on the basis of this information. The starting point for a reconfiguration of a layer l is therefore the expression

$$\texttt{let (s}_{\texttt{init}}\texttt{,hdlr) = convert } l \texttt{ args (ls, vs)} \quad \texttt{in} \quad \texttt{hdlr(}s_l\texttt{, } \mathit{event}\texttt{)}$$

where $\texttt{s}_{\texttt{init}}$ describes the initial state of l, `hdlr` its event handler, s_l its current state, and *event* an incoming event of the form `UpM(ev,hdr)` or `DnM(ev,hdr)`. Assumptions about the common case are usually stated in the form of equations which characterize the type of the event `ev` (sending or broadcasting), the structure of the header `hdr` (full header, local header, no new information), and the current state s_l. No assumptions will be made about the local and global view states (`ls,vs`) or the additional arguments `args` used in the definition of the layer. For the sake of clarity we have defined a NuPRL-abstraction for the above expression that also binds the free variables, and a display form that allows us to use the following notation in a formal reconfiguration.

```
RECONFIGURE LAYER l
  FOR EVENT event
  AND STATE s_l
ASSUMING    assumptions
```

```
let convert l args view =
  let up ev hdr (s,q)  = (s, Fqueue.add (UpM(ev,hdr)) q)
  and dn ev hdr (s,q)  = (s, Fqueue.add (DnM(ev,hdr)) q)
  and s,h              = l args view                          in

  let {up_lin=up;dn_lin=dn} = h {up_lout=up;dn_lout=dn}       in

  let hdlr (s,ev) = match ev with
      | UpM(ev,hdr) -> up ev hdr (s, Fqueue.empty)
      | DnM(ev,hdr) -> dn ev hdr (s, Fqueue.empty)            in

  (s,hdlr)
```

Fig. 3. Conversion function for a functional implementation of layers

Regardless of the chosen layer l and the assumptions about the common case, a reconfiguration always begins with a fixed series of symbolic evaluation steps for function in-lining and reduction of abstractions. First, we unfold the application of `convert` (see Figure 3) to l `args (ls,vs)`, then l `args (ls,vs)`, and finally `Layer.hdr init`$_l$ `hdlrs`$_l$ `args (ls,vs)`. Afterwards we evaluate the resulting expression as far as possible, using our evaluation tactics for OCAML-expressions. As the code for `hdlrs`$_l$ has a fixed structure (see Figure 2) we evaluate it until the let abstractions `up_hdlr`, `uplm_hdlr`, `upnm_hdlr`, `dn_hdlr`, `dnnm_hdlr` have been unfolded. Finally we analyze the outer structure of the event (`UpM(ev,hdr)` or `DnM(ev,hdr)`) and the structure of the header (`Full(hdr,abv), Local hdr`, or `NoMsg`), which leads to further reductions.

All these steps could be executed by a general reduction tactic called `Red`, which is based on the symbolic evaluation tactics for OCAML-expressions. But it would be inefficient to do so, as `Red` must search for reducible sub-expressions, although the precise order of reductions and the position of the reducible sub-expressions in the term-tree is fixed. We have therefore written a specialized tactic `RedLayerStructure`, which performs these steps in the most efficient order and will always be executed at the beginning of a reconfiguration.

The steps so far are nothing but controlled code optimizations that can be considered as special case of partial evaluation and have little to do with the fast-path through the layer. In the following steps make use of the assumptions. Since these are usually expressed as equations or can be converted into a list of equations by formal reasoning, we can apply equality substitutions for each assumption and then evaluate the piece of code that was affected by a substitution. Again, we use a specialized tactic `UseHyps` for this purpose.[1] Usually the assumption about the type of the event `ev` (`ESend` or `ECast`) are used first, while the assumptions about the current state are substituted when they can be used to eliminate branches of a conditional or a case expression.

After these steps further reductions may be possible until the resulting code for the fast-path is in acceptable form (it is not meaningful to evaluate the code until it contains only basic OCAML-expressions). Therefore a user will apply the tactic `Red` as long as meaningful progress can be made and finally call the tactic `Derivation_Completed` to indicate the completion of the reconfiguration.

[1] `UseHyps` replaces inequalities by `true` or `false`, even if the code contains a variation of a given inequality, e.g. `b>a`, `not(a>=b)`, or `not(b<=a)` instead of `a<b`.

Example 1. We illustrate the reconfiguration of the `Bottom` layer for received broadcast messages. These messages have the form `UpM(ev,Full(header,hdr))` where (1) `ev` is a broadcast event (`getType ev = ECast`), (2) there is no header (`header = Bottom.NoHdr`), and (3) the state `s_bottom` does not record a previous failure of the sender (`s_bottom.failed.(getPeer ev) = false`). We start with

```
⊢ RECONFIGURE LAYER Bottom
     FOR EVENT UpM(ev, Full(NoHdr, hdr))
     AND STATE s_bottom
   ASSUMING    getType ev = ECast ∧ s_bottom.failed.(getPeer ev) = false
```

Applying `RedLayerStructure` decomposes the above expression, which declares `s_bottom, ev, hdr` as free variables and makes the assumptions explicit, and then evaluates the remaining program expression as described above.

```
GIVEN s_bottom, ev, hdr
ASSUME
  4. getType ev = ECast
  5. s_bottom.failed.(getPeer ev) = false
⊢ (match (getType ev, NoHdr) with
       ((ECast | ESend), NoHdr)  -> ......
     | (ECast, Unrel)            -> ......
     | (ESend, Unrel)            -> ......
     | (_, MergeRequest)         -> ......
     | (_, MergeGranted)         -> ......
     | (_, MergeDenied)          -> ......
     | _                         -> ......
  )  (s_bottom, Fqueue.empty)
```

where '`......`' indicates that details of the code are temporarily hidden from the display. We now call `UseHyps [4]`, which leads to an evaluation of the first case of the case expression while all the other cases are eliminated from the code.

```
⊢ (if s_bottom.all_alive or (not (s_bottom.failed.(getPeer ev)))
    then fun ((s, q))  -> (s, Fqueue.add UpM(ev, hdr) q)
    else free name ev
  ) (s_bottom, Fqueue.empty)
```

At this point using the assumption `s_bottom.failed.(getPeer ev) = false` leads to an evaluation of the first case of the conditional. As further general reductions are possible, we use the tactic `UseHyps [5] THEN RepeatFor 4 Red` to get the following piece of code

```
⊢ (s_bottom, Fqueue.add UpM(ev,hdr) Fqueue.empty)
```

No further reductions are meaningful, as the resulting state `s_bottom` and the queue of outgoing events (a queue containing the single event `UpM(ev,hdr)`)[2] are explicitly stated. Under the the given assumptions we know now

$$\texttt{hdlr}_b\texttt{(s\_bottom, UpM(ev, Full(NoHdr,hdr)))} \;=\; \texttt{(s\_bottom,[:UpM(ev,hdr):])}$$

where $\texttt{hdlr}_b$ denotes the event handler of the bottom layer. This means that the layer's state remains unchanged while the original message is passed to the next layer after the header `NoHdr` has been stripped off. This completes the derivation.

[2] $\texttt{[: ev}_1\texttt{; ...; ev}_n\texttt{ :]}$ abbreviates the expression
$\texttt{Fqueue.add ev}_1\texttt{ (Fqueue.add ... (Fqueue.add ev}_n\texttt{ Fqueue.empty) ..)}$

By combining the tactics `RedLayerStructure`, `UseHyps`, and `Red` a reconfiguration of a protocol layer can be performed almost automatically. So far we have successfully used them to reconfigure 25 of ENSEMBLE's 40 protocol layers for the 4 most common kinds of events, i.e. up- and down-going send and broadcast events. In many cases a layer consisting of 300–500 lines of code is reduced to a simple update of the state and a single event to be passed to the next layer.

2.2 Verifying a reconfiguration

A fast-track reconfiguration in NUPRL is more than just a syntactical transformation of program code. Since it is based entirely on substitution and evaluation mechanisms we *know* that, under the given assumptions, a reconfigured program is equivalent to the original one. But in order to *guarantee* the reliability of a reconfigured communication system we must provide a formal proof of this equivalence. Together with a verification of the original "reference" implementation (which, although by no means trivial, is much easier than verifying the optimized code) this proof will lead to a verification of an efficient system that will actually be used in safety-critical applications. Formally, we have to prove the equality

$$\texttt{let (s}_{\texttt{init}}\texttt{,hdlr) = convert } l \texttt{ args (ls, vs)} \quad \texttt{in} \quad \texttt{hdlr(}s_l\texttt{,}\mathit{event}\texttt{)}$$
$$= \quad (s_l', \ \texttt{[:}\mathit{out\text{-}events}\texttt{:]})$$

where the left equand is the starting point of a reconfiguration and the right equand its final result, consisting of a modified state s_l' and a queue of outgoing events `[:`*out-events*`:]`. Again we introduce a formal abbreviation:

```
RECONFIGURE LAYER l
   FOR EVENT  event
   AND STATE  s_l
   ASSUMING   assumptions
YIELDS EVENTS [:out-events:]
   AND STATE  s_l'
```

Fortunately, there is a close correspondence between the substitution and evaluation mechanisms that we have used for reconfigurations and the logical inference rules of the NUPRL proof development system. In fact, it is quite easy to write *proof* tactics which perform exactly the same steps on the left hand side of an equation as our *reconfiguration* tactics did on the code of the protocol layer. We can therefore consider the trace of a reconfiguration as plan for the equivalence proof and transform each reconfiguration step into the corresponding proof step. This makes it possible to prove the equivalence theorem completely automatically – even in cases where the reconfiguration required user interaction.

We have written a tactic `CreateReconfVerify`, which states the equivalence theorem and proves it to be correct by replaying the derivation of the reconfigured code. No search is involved in the process and the tactic is guaranteed to succeed. We can run it as background process as soon as a reconfiguration has been finished. By calling the transformation tactic `DONE` as final step of a reconfiguration a user may initiate an immediate verification of the reconfigured code (this includes a final call of the tactic `Derivation_Completed`). Executing this tactic as final step in our reconfiguration of the `Bottom` layer in example 1,

```
⊢ ∀s_bottom:Bottom.state. ∀ev:Event.t. ∀hdr:Headers.
|  RECONFIGURING LAYER Bottom
|    FOR EVENT  UpM(ev, Full(NoHdr, hdr))
|    AND STATE  s_bottom
|    ASSUMING   getType ev = ECast  ∧  s_bottom.failed.(getPeer ev) = false
|  YIELDS EVENTS [:UpM(ev, hdr):]
|     AND STATE  s_bottom
|
BY InitReconf THEN OnFirstEquandRedLayerStructure
\
 GIVEN s_bottom, ev, hdr
 ASSUME
   4. getType ev = ECast
   5. s_bottom.failed.(getPeer ev) = false
 ⊢ (match (getType ev, NoHdr) with
 |         ((ECast | ESend), NoHdr)  -> .........
 |       | (ECast, Unrel)            -> .........
 |       | (ESend, Unrel)            -> .........
 |       | (_, MergeRequest)         -> .........
 |       | (_, MergeGranted)         -> .........
 |       | (_, MergeDenied)          -> .........
 |       | _                         -> .........
 |    )  (s_bottom, Fqueue.empty)
 |   = (s_bottom, [:UpM(ev, hdr):])
 |
 BY OnFirstEquandUseHyps [4]
 \
  ⊢ (if s_bottom.all_alive or (not (s_bottom.failed.(getPeer ev)))
  |      then fun ((s, q))  -> (s, Fqueue.add UpM(ev, hdr) q)
  |      else free name ev
  |    ) (s_bottom, Fqueue.empty)
  |   = (s_bottom, [:UpM(ev, hdr):])
  |
  BY OnFirstEquandUseHyps [5] THEN RepeatFor 4 Red
  \
   ⊢ (s_bottom, Fqueue.add UpM(ev, hdr) Fqueue.empty)
   |   = (s_bottom, [:UpM(ev, hdr):])
   |
   BY Fold `Fqueue.expression`
   \
    ⊢ (s_bottom, [:UpM(ev, hdr):]) = (s_bottom, [:UpM(ev, hdr):])
    |
    BY Equality
```

Fig. 4. Reconfiguration theorem for the `Bottom` layer and up-going broadcast events

for instance, creates the verification presented in figure 4. Alternatively one may complete a reconfiguration without verifying it and cause the system to create verifications of all reconfigurations off-line at the end of a NuPRL session.

The automatic verification of code reconfigurations does not only prove the equivalence between original and reconfigured code under the given assumptions but also allows us to use more efficient reconfiguration mechanism. Since we generate a formal equivalence proof anyway, it is not necessary to implement the reconfiguration tactics on the basis of NuPRL's elementary inference rules. Instead, we can *calculate* the result of each reconfiguration step by consulting redex-contracta tables for OCAML-redices and transform the code of a protocol layer entirely on the basis of these calculations, using NuPRL's editor only as a display mechanism for the necessary interactions. Compared to an evaluation of the type-theoretical expressions that represent the code until they are in the desired form this technique drastically reduces the overhead of program transformations. Experiments have shown that bypassing NuPRL's inference system *during a reconfiguration* speeds up the response times of our reconfiguration tactics by at least a factor of five[3] and thus makes a reconfiguration process that involves user interaction practically feasible.

```
(* SPLIT: (up,dn) q adds the up and dn events to the up and dn queues   *
 *                 and returns the resulting queues.                     *)
   let split updn b =
     Fqueue.fold (fun (up,dn) ev ->
       match ev with
       | UpM(ev,hdr) -> let up = Fqueue.add (UpM(ev,hdr)) up in  (up,dn)
       | DnM(ev,hdr) -> let dn = Fqueue.add (DnM(ev,hdr)) dn in  (up,dn)
     ) updn b

(* SPLIT_TOP: splits the output events of the upper layer *)
   let split_top emit_midl q = split emit_midl q

(* SPLIT_BOT: Same as split_top except this is for the bottom layer, so  *
            emit and midl get reversed before/after the call to SPLIT. *)
   let split_bot (emit,midl) q =
     let midl,emit = split (midl,emit) q in   (emit,midl)

(* A totally functional layer composition function   *)
   let compose top bot state vf =
     let s1,top_hdlr = top state vf in      (* Initialize the two layers *)
     let s2,bot_hdlr = bot state vf in
     let loop (s1,s2) (emit,midl) =
       Fqueue.loop (fun ((s1,s2),emit) midl ev ->
         match ev with
         | UpM(ev,hdr) -> let s1,evs = top_hdlr (s1,UpM(ev,hdr)) in
                          let (emit,midl) = split_top (emit,midl) evs in
                           (((s1,s2),emit),midl)

         | DnM(ev,hdr) -> let s2,evs = bot_hdlr (s2,DnM(ev,hdr)) in
                          let (emit,midl) = split_bot (emit,midl) evs in
                           (((s1,s2),emit),midl)

       ) ((s1,s2),emit) midl
     in
                  (* Outer handler takes a single event and then passes it to *
                   * appropriate layer and then splits the emitted events.    *)
     let hdlr = function
       | ((s1,s2),DnM(ev,hdr)) ->
            let s1,emitted = top_hdlr (s1,DnM(ev,hdr)) in
             let (emit,midl) = split_top (Fqueue.empty,Fqueue.empty) emitted in
              loop (s1,s2) (emit,midl)
       | ((s1,s2),UpM(ev,hdr)) ->
            let s2,emitted = bot_hdlr (s2,UpM(ev,hdr)) in
             let (emit,midl) = split_bot (Fqueue.empty,Fqueue.empty) emitted in
              loop (s1,s2) (emit,midl)
     in
        ((s1,s2),hdlr)
```

Fig. 5. Ensemble code for composing two protocol layers

3 Protocol Stack Reconfiguration

In contrast to the code of the individual protocol layers, application protocol stacks have no a priori implementation but are defined according to the demands of the application system. As there are thousands of possible configurations, a system expert who designs an application system on the basis of the Ensemble group communication toolkit must also be given a tool that allows him to create a fast-track reconfiguration of the system almost completely automatically.

Protocol stacks are generated by layer composition. The function `compose`, whose functional implementation is presented in Figure 5, glues two converted layers together. The initial state of the resulting layer is the pair of the two

[3] On a 75MHz Sparc 20 running the Lucid Common Lisp implementation of NuPRL each reconfiguration step operating on layers with 200–500 lines of code takes now between 10 and 50 seconds. The response times for a 170Mhz UltraSparc or a 166 Mhz Pentium running NuPRL under Allegro Common Lisp are about the same.

initial states. The new event handler `hdlr` has to deal with up-going events (`UpM(ev,hdr)`) that enter the lower layer or down-going events (`DnM(ev,hdr)`) that enter the upper layer. As the layers may emit both up- and down-going events, `hdlr` must distinguish events that are passed to the respective other layer from those that leave the stack. In principle, it is even possible that events bounce between the two layers. Therefore the function `compose` must rely on unrestricted recursion, i.e. on the function `Fqueue.loop`, to iterate the sending of events until the queue `midl` of events that pass between the layers is empty.

Correspondingly, a reconfiguration of protocol stacks has to proceed recursively while tracing the path of common events through the stack. In a two-layer stack, for instance, up-going events are first passed to the lower layer. After the code of this layer has been reconfigured accordingly, all generated down-going events are stored in the queue of events to be emitted while up-going events will be passed to the upper layer. For these events the code of the upper layer will be reconfigured and the generated events will again be split into up-events to be emitted and down-events to be passed to the lower layer. The reconfiguration process continues until no more events pass between the layers. Obviously, it only 'succeeds' if the assumptions do in fact allow a simplification of the code. Otherwise the generated code would be more complex than the original one.

Although it is possible to write a reconfiguration tactic for protocol stacks that performs these steps in the most efficient order, the tactic-based approach to reconfiguration, which was successful for the reconfiguration of individual protocol layers, does not scale up very well. Even for small protocol stacks a tactic-based reconfiguration involves many rewrite steps and has to deal with extremely large terms. For each layer it has to analyze the complete code whenever an event is being passed to this layer. Tracing events through the code of a realistic protocol stack is therefore extremely time-consuming, since we only know the structure of the events and have to rely on *symbolic* evaluation.[4]

On the other hand, the characteristic property of a *fast-track* is that in the common case an event passes straight through a protocol stack without generating more than one or two additional events. In these cases one may logically *derive* the result of reconfiguring a protocol stack from the results of reconfiguring its individual protocol layers. Instead of symbolically evaluating the code of the protocol stack one only has to compose these results according to our knowledge about the common effects of layer composition.

Formally, this can be achieved by composing generic theorems about the reconfiguration of individual protocol layers, using theorems about the result of layer composition for simple event traces. The latter will serve as *derived inference rules* on a very high level of abstraction, as they describe how to compose arbitrary protocol layers in a *single* inference step where a tactic-based reconfiguration would have to execute thousands of elementary steps.

This methodology, illustrated in Figure 6, does not only lead to a better performance of the reconfiguration process but also a much clearer style of reasoning.

[4] Given a reconfiguration time of at least one minute per layer the response time between necessary user interactions becomes unacceptable.

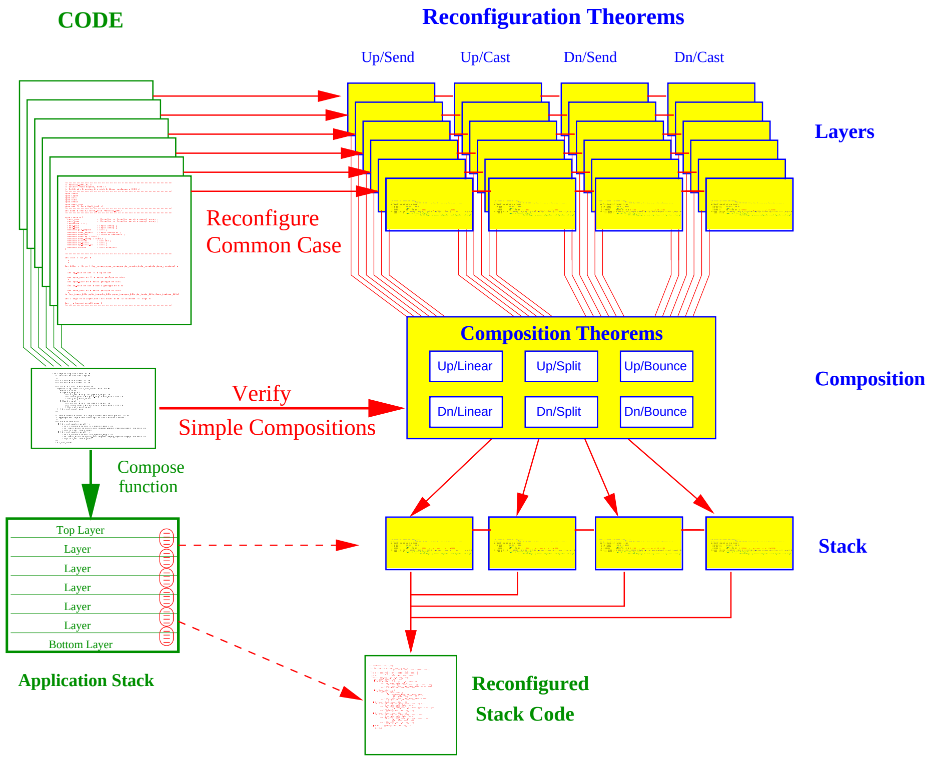

Fig. 6. Reconfiguration methodology: composing reconfiguration theorems

Furthermore, system updates can be handled much easier: the modification of a layer's code usually only requires reproving the reconfiguration theorems for this particular layer while the reconfiguration of the protocol stack will remain unaffected or can be re-executed automatically.

As generic *reconfiguration theorems* for individual protocol layers we can use the equivalence theorems discussed in section 2.2. For each of ENSEMBLE's protocol layers layer we have to create a reconfiguration theorem for up- and down-going events, sending and broadcasting, which are the most common cases during a communication. The proofs of these theorems require a certain amount of user interaction. But they can be done once and for all for each release of the ENSEMBLE toolkit and will be included in the distribution.

Composition theorems formulate the logical laws of layer composition and deal with the common cases of fast-paths through a protocol stack, such as linear traces, bouncing events, and events that cause several events to be emitted from a layer (splitting) – both for up- and down-going events. As they have to reason about all possible stacks they must necessarily be higher-order theorems.

```
THM ComposeDnLinear
 ∀Upper,Lower,s_lower,s1_lower,s_upper,s1_upper, ev, hdr, hdr1, hdr2
    RECONFIGURING LAYER Upper FOR EVENT     DnM(ev, hdr)     AND STATE s_upper
                             YIELDS EVENTS [:DnM(ev, hdr1):] AND STATE s1_upper
  ∧ RECONFIGURING LAYER Lower FOR EVENT     DnM(ev, hdr1)    AND STATE s_lower
                             YIELDS EVENTS [:DnM(ev, hdr2):] AND STATE s1_lower
  ⇒ RECONFIGURING LAYER compose Upper Lower
         FOR EVENT     DnM(ev, hdr)     AND STATE (s_upper, s_lower)
      YIELDS EVENTS [:DnM(ev, hdr2):] AND STATE (s1_upper, s1_lower)
```

Fig. 7. Reconfiguration theorem for linear down traces

```
THM ComposeUpSplit
 ∀Upper,Lower,s_lower,s1_lower,s2_lower,s_upper,s1_upper, ev, hdr, hdr1, hdr2, hdr3, hdr4
   RECONFIGURING LAYER Lower FOR EVENT    UpM(ev, hdr)     AND STATE s_lower
                             YIELDS EVENTS [:UpM(ev, hdr1):] AND STATE s1_lower
 ∧ RECONFIGURING LAYER Upper FOR EVENT    UpM(ev, hdr1)    AND STATE s_upper
                             YIELDS EVENTS [:UpM(ev, hdr2); DnM(ev, hdr3):] AND STATE s1_upper
 ∧ RECONFIGURING LAYER Lower FOR EVENT    DnM(ev, hdr3)    AND STATE s1_lower
                             YIELDS EVENTS [:DnM(ev, hdr4):] AND STATE s2_lower
  ⇒ RECONFIGURING LAYER compose Upper Lower
        FOR EVENT    UpM(ev, hdr)                  AND STATE (s_upper, s_lower)
     YIELDS EVENTS [:DnM(ev, hdr4);UpM(ev, hdr2):] AND STATE (s1_upper, s2_lower)
```

Fig. 8. Reconfiguration theorem for up traces with event splitting

Figure 7 presents a reconfiguration theorem for composing fast-tracks for down-going *linear traces*, where an event passes through the stack without generating additional events. In this case each layer `Upper` and `Lower` yields a queue consisting of a single down-event and the composition of both does the obvious. While the statement of this theorem is almost trivial, its proof is rather complex as we have to reason about the actual code of ENSEMBLE's `compose` function and bout the result of all the steps that would usually be executed *during* a reconfiguration. Thus by proving this theorem we remove the deductive burden from the reconfiguration process itself. Figure 8 shows a similar theorem for up-going events with event splitting in the upper layer.

The logical reconfiguration of protocol stacks on the basis of these theorems proceeds in two stages. In the first stage we generate the statement of a reconfiguration theorem by calling `ReconfStackGoal layers dir EType`, where `layers` denotes the names of the protocol layers in the stack, `dir` denotes the direction of the incoming event, and `EType` its type, i.e. sending or broadcasting. `ReconfStackGoal` composes the reconfiguration theorems for the individual protocol layers according to the information provided in the composition theorems. By matching incoming and outgoing events in the theorems for adjacent layers it determines the structure of the events that enter the stack and the result of passing them through it. The states of the layers will essentially be composed into tuples of states and the assumptions will be accumulated by conjunctions.

Example 2. To generate a reconfiguration theorem for the three-layer stack `Pt2pt::Mnak::Bottom`[5] for down-going send-events, `ReconfStackGoal` consults the following three reconfiguration theorems in the library.

```
THM Pt2ptReconfDnMESend_verif
 ∀s_pt2pt:Pt2pt.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
 RECONFIGURING LAYER Pt2pt
    FOR EVENT  DnM(ev, hdr)
    AND STATE  s_pt2pt
    ASSUMING   getType ev = ESend   ∧   getPeer ev ≠ ls.rank
 YIELDS EVENTS [:DnM(ev,Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),hdr)):]
    AND STATE  s_pt2pt[.sends.(getPeer ev) ← Iq.add s_pt2pt.sends.(getPeer ev) (getIov ev) hdr]

THM MnakReconfDnMESend_verif
 ∀s_mnak:Mnak.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
 RECONFIGURING LAYER Mnak
    FOR EVENT  DnM(ev, hdr)
    AND STATE  s_mnak
    ASSUMING   getType ev = ESend
 YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
    AND STATE  s_mnak
```

[5] The infix notation `Upper::Lower` abbreviates `compose Upper Lower`

```
THM BottomReconfDnMESend_verif
 ∀s_bottom:Bottom.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
 RECONFIGURING LAYER Bottom
   FOR EVENT    DnM(ev, hdr)
   AND STATE    s_bottom
   ASSUMING     getType ev = ESend   ∧   s_bottom.enabled
 YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
    AND STATE   s_bottom
```

Since all three layers show a linear behavior for down-going send-events, we have to compose them in a way that makes the theorem `ComposeDnLinear` applicable. The incoming event of theorem `Pt2ptReconfDnMESend_verif` describes the event that enter the three-layer stack. The outgoing event of `Pt2pt` has to be matched against the incoming event of `Mnak` and the variable `hdr` in theorem `MnakReconfDnMESend_verif` must be instantiated accordingly. Similarly, the outgoing event of `Mnak` will be matched against the incoming event of `Bottom`. The instantiated outgoing event of theorem `BottomReconfDnMESend_verif` describes the event queue emitted by the stack `Pt2pt::Mnak::Bottom`. The initial and resulting states of the three (instantiated) theorems are composed into triples and while the assumptions are composed by conjunction. As a result we get the following reconfiguration theorem for the stack `Pt2pt::Mnak::Bottom`.

```
THM ReconfStackDnMESend_Pt2ptMnakBottom
∀s_pt2pt:Pt2pt.state.∀s_mnak:Mnak.state.∀s_bottom:Bottom.state.∀ls,ev,hdr
 RECONFIGURING LAYER Pt2pt::Mnak::Bottom
    FOR EVENT   DnM(ev, hdr)
    AND STATE   (s_pt2pt, s_mnak, s_bottom)
    ASSUMING    getType ev = ESend  ∧  getPeer ev ≠ ls.rank  ∧  s_bottom.enabled
 YIELDS EVENTS [:DnM(ev, Full(NoHdr, Full(NoHdr,
                                        Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)), hdr)):]
    AND STATE   ( s_pt2pt[.sends.(getPeer ev) ← Iq.add s_pt2pt.sends.(getPeer ev) (getIov ev) hdr]
                , s_mnak
                , s_bottom
                )
```

In the second phase we have to create a formal proof of the generated reconfiguration theorem. As this proof will essentially rely on composition theorems and the reconfiguration theorems for the individual protocol layers we can re-use much of the information that we gained while stating the theorem. After initializing the proof, i.e. declaring variables and making the assumptions explicit, we insert instantiated versions of the relevant layer reconfiguration theorems as additional hypotheses into the proof, using the tactic `QuoteLayerVerifs`. By repeatedly calling the tactic `ComposeReconfigurations` we then compose these theorems step by step according to the composition theorems. This eventually creates a hypothesis that is identical to the original statement of the theorem. Calling the elementary inference rule `hyp` we compare this hypothesis with the conclusion and thus complete the proof. An example proof based on this methodology is presented in figure 9.

To automate the reconfiguration of protocol stacks we have written a meta-level function `CreateStack` that generates the reconfiguration theorem, proves it correct, and stores it under a unique name. In contrast to the purely tactic-based approach to protocol stack verification it does not require user interaction and can thus be used as reconfiguration tool by people who have no further knowledge of NuPRL. Furthermore, it performs its task in *linear time* with respect to the number of protocol layers that have to be passed by events, as each layer

```
⊢∀s_pt2pt:Pt2pt.state.∀s_mnak:Mnak.state.∀s_bottom:Bottom.state.∀ls,ev,hdr
| RECONFIGURING LAYER Pt2pt::Mnak::Bottom
|    FOR EVENT  DnM(ev, hdr)      AND STATE (s_pt2pt, s_mnak, s_bottom)
|    ASSUMING   getType ev = ESend  ∧  getPeer ev ≠ ls.rank  ∧  s_bottom.enabled
| YIELDS EVENTS [:DnM(ev, Full(NoHdr, Full(NoHdr,
|                             Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),hdr)))):]
|    AND STATE  (s_pt2pt[.sends.(getPeer ev) ← Iq.add s_pt2pt.sends.(getPeer ev)(getIov ev) hdr]
|               , s_mnak
|               , s_bottom
|               )
|
BY InitStackReconf
\
 GIVEN s_pt2pt, s_mnak, s_bottom, ev, hdr.
 ASSUME
   6.getType ev = ECast
   7.getPeer ev ≠ ls.rank
   8.s_bottom.enabled
 ⊢ RECONFIGURING LAYER Pt2pt::Mnak::Bottom
 |     FOR EVENT DnM(ev, hdr)      AND STATE (s_pt2pt, s_mnak, s_bottom)
 | YIELDS EVENTS [:DnM(ev, Full(NoHdr, Full(NoHdr,
 |                             Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),hdr)))):]
 |    AND STATE  (s_pt2pt[.sends.(getPeer ev) ← Iq.add s_pt2pt.sends.(getPeer ev)(getIov ev) hdr]
 |               , s_mnak
 |               , s_bottom
 |               )
 |
 BY QuoteLayerVerifs
 \
  ASSUME
   9.RECONFIGURING LAYER Pt2pt
  |      FOR EVENT DnM(ev, hdr)      AND STATE s_pt2pt
  |  YIELDS EVENTS [:DnM(ev, Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)), hdr)):]
  |     AND STATE  s_pt2pt[.sends.(getPeer ev) ← Iq.add s_pt2pt.sends.(getPeer ev)(getIov ev) hdr]
  10.RECONFIGURING LAYER Mnak
  |     FOR EVENT  DnM((ev, Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),hdr)):]
  |     AND STATE  s_mnak
  |  YIELDS EVENTS [:DnM(ev, Full(NoHdr, Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),hdr))):]
  |     AND STATE  s_mnak
  11.RECONFIGURING LAYER Bottom
  |     FOR EVENT  DnM(ev, Full(NoHdr, Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)), hdr)))
  |     AND STATE  s_bottom
  |  YIELDS EVENTS [:DnM(ev, Full(NoHdr, Full(NoHdr,
  |                               Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),hdr)))):]
  |     AND STATE  s_bottom
  ⊢ RECONFIGURING LAYER Pt2pt::Mnak::Bottom
  |    FOR EVENT  DnM(ev, hdr)      AND STATE (s_pt2pt, s_mnak, s_bottom)
  | YIELDS EVENTS [:DnM(ev, Full(NoHdr, Full(NoHdr,
  |                             Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),hdr)))):]
  |    AND STATE  (s_pt2pt[.sends.(getPeer ev) ← Iq.add s_pt2pt.sends.(getPeer ev)(getIov ev) hdr]
  |               , s_mnak
  |               , s_bottom
  |               )
  |
  BY ComposeReconfigurations
  \
   ASSUME
   12.RECONFIGURING LAYER  Mnak::Bottom
   |     FOR EVENT  DnM((ev, Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)), hdr)):]
   |     AND STATE  (s_mnak, s_bottom)
   |  YIELDS EVENTS [:DnM(ev, Full(NoHdr, Full(NoHdr,
   |                               Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),hdr)))):]
   |     AND STATE (s_mnak,s_bottom)
   ⊢ RECONFIGURING LAYER Pt2pt::Mnak::Bottom
   |    FOR EVENT  DnM(ev, hdr)      AND STATE (s_pt2pt, s_mnak, s_bottom)
   | YIELDS EVENTS [:DnM(ev, Full(NoHdr, Full(NoHdr,
   |                             Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),hdr)))):]
   |    AND STATE  (s_pt2pt[.sends.(getPeer ev) ← Iq.add s_pt2pt.sends.(getPeer ev)(getIov ev) hdr]
   |               , s_mnak
   |               , s_bottom
   |               )
   |
   BY ComposeReconfigurations  THEN  hyp (-1)
```

Fig. 9. Proof of theorem `ReconfStackDnMESend_Pt2ptMnakBottom` from example 2

essentially requires only two inference steps (instantiating its reconfiguration theorem and composing it with the rest of the stack) in the derivation. As a consequence, our theorem-based reconfiguration method scales up very well and can be used for an efficient reconfiguration of applications protocol stacks.

4 Header Compression

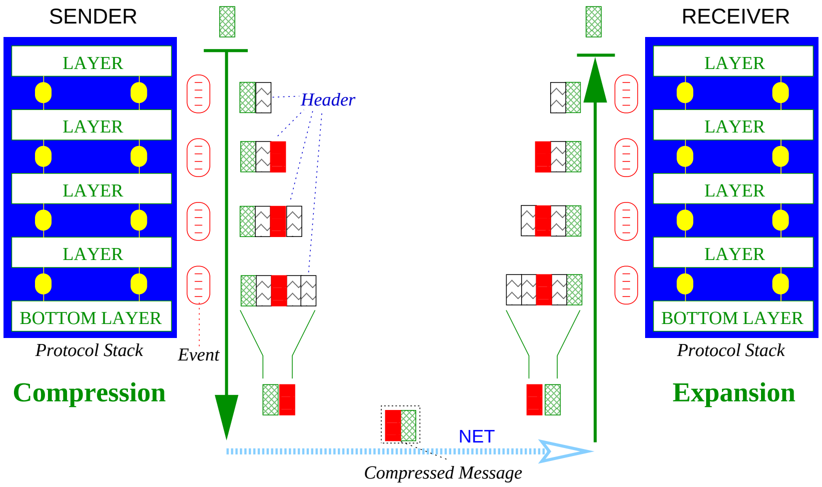

Fig. 10. Header Compression: superfluous headers are suppressed

The fast-track reconfiguration of protocol stacks leads already to a significant speed-up of the communication software, as it improves the efficiency of the protocol stacks. In the common case, as it turned out, most protocol layers are not activated while an event is being passed. They only add a header `NoHdr` to a down-going event, which indicates the fact that the corresponding layer on the way up does not have to be activated as well. Since this behavior is identical for all common events of the same kind, it is not really necessary to transmit these "no-information" headers over the net. Instead, it is worth while to *compress* a header before it is sent and to *expand* it again into its original form before it enters the receiver's stack. This will reduce the net load and improve the overall efficiency of group communication.

Header compression for common events can easily be generated after a fast-track reconfiguration, since the reconfigured code clearly states which layers have been activated. The reconfiguration theorem for down-going send events through the stack `Pt2pt::Mnak::Bottom` in example 2, for instance, states that only the `Pt2pt` layer has been active. The header

```
Full(NoHdr, Full(NoHdr, Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)), hdr)))
```

can therefore be compressed to

```
OptSend(Iq.hi s_pt2pt.sends.(getPeer ev), hdr):]
```

because the keywords `Full`, `NoHdr`, and `Data` contain no essential information. Similarly we can compress common down-going cast-events while all other events will not be affected by compression. Message expansion simply inverts compression and can easily be programmed with `Ocaml`'s pattern matching features.

```
let wrap_hdr compress expand stack state vs =
 let (s,h)       = stack state vs in
 let emit (s,e)  = let e1 = Fqueue.map (function
                                 | UpM(ev,hdr) -> UpM(ev,hdr)
                                 | DnM(ev,hdr) -> DnM(ev,compress hdr)
                                 ) e
                  in  (s,e1)
 in
 let hdlr (s,ev) = match ev with
                   | UpM(ev,hdr) -> emit (h (s,(UpM(ev,expand hdr))))
                   | DnM(ev,hdr) -> emit (h (s,(DnM(ev,hdr))))
 in
    (s,hdlr)
```

Fig. 11. Ensemble-code for wrapping layers with compression/expansion

We have written a meta-level function `CreateCompressionModule` which consults the reconfiguration theorems for a given stack and automatically generates a small compression/expansion module for this stack.

Example 3. `CreateCompressionModule [‘Pt2pt‘;‘Mnak‘;‘Bottom‘]` creates the following Ocaml module for the stack `Pt2pt::Mnak::Bottom`.

```
open Event

type ('a, 'b) msg_opt =   Normal of a
                        | OptSend of Trans.seqno * b
                        | OptCast of Trans.seqno * b

let compress hdr = match hdr with
    Full(NoHdr, Full(NoHdr, Full(Data(seqno), hdr))) -> OptSend(seqno, hdr)
  | Full(NoHdr, Full(Data(seqno), Full(NoHdr, hdr))) -> OptCast(seqno, hdr)
  | hdr                                               -> Normal(hdr)

let expand hdr = match hdr with
    OptSend(seqno, hdr)  -> Full(NoHdr, Full(NoHdr, Full(Data(seqno), hdr)))
  | OptCast(seqno, hdr)  -> Full(NoHdr, Full(Data(seqno), Full(NoHdr, hdr)))
  | Normal(hdr)          -> hdr
```

Given two functions `compress` and `expand`, the function `wrap_hdr`, presented in figure 11, wraps the protocol stack with header compression. Emitted downgoing events will be compressed before they leave the stack while incoming events will be expanded before they enter. Since `compress` and `expand` are fixed for a given protocol stack, we will formally abbreviate the wrapped stack `wrap_hdr compress expand` *`stack`* by *`stack`* `WRAPPED WITH COMPRESSION`.

Header compression is applied to all events passing through a protocol stack, even to those which are not "common" events. But it becomes particularly efficient when combined with fast-track reconfigurations, as all events on the fast-path will also be affected by compression and expansion. Reconfiguring stacks that are wrapped with compression will therefore lead to a greater efficiency, as the resulting code will directly operate on the compressed messages. The logical laws that allow us to combine reconfiguration with header compression are expressed by the *compression/expansion theorems* in figure 12.

A logical reconfiguration of wrapped protocol stacks follows the same methodology as before. In a first stage we generate the statement of a reconfiguration theorem by calling `ReconfCompressGoal layers dir EType`. This composes the reconfiguration theorems for the individual protocol layers and adds header

```
THM Compress
  ∀L,s,s1,ev,hdr,hdr1,compress,expand
    RECONFIGURING LAYER L
     FOR EVENT        DnM(ev, hdr)     AND STATE s
     YIELDS EVENTS [:DnM(ev, hdr1):] AND STATE s1
⇒  RECONFIGURING LAYER L   WRAPPED WITH COMPRESSION
     FOR EVENT        DnM(ev, hdr)              AND STATE s
     YIELDS EVENTS [:DnM(ev, compress hdr1):] AND STATE s1

THM Expand
  ∀L,s,s1,ev,hdr,hdr1,compress,expand
    RECONFIGURING LAYER L
     FOR EVENT        UpM(ev, expand hdr)  AND STATE s
     YIELDS EVENTS [:UpM(ev, hdr1):]       AND STATE s1
⇒  RECONFIGURING LAYER L   WRAPPED WITH COMPRESSION
     FOR EVENT        UpM(ev, hdr)     AND STATE s
     YIELDS EVENTS [:UpM(ev, hdr1):] AND STATE s1
```

Fig. 12. Compression and expansion theorems for down/up-traces

compression and expansion according to the theorems `Compress` and `Expand` in figure 12. To prove this statement, we first create a hypothesis about the result of a regular fast-track reconfiguration and apply the tactics `QuoteLayerVerifs` and `ComposeReconfigurations`. For down-going events we then transform emitted events in the conclusion into the form `compress` ***`hdr`*** and finally apply the theorem `Compress`. For up-going events we transform the input into the form `expand` ***`hdr`*** and apply the theorem `Expand`.

All these steps – generating the compression module, stating the reconfiguration theorem for the wrapped stack, and proving it it correct – can be performed fully automatically by a meta-level function `CreateCompressedStack`. For down-going send-events through the stack `Pt2pt::Mnak::Bottom` with compression, for instance, this function creates the following reconfiguration theorem.

```
THM ReconfCompressedDnMESend_Pt2ptMnakBottom
∀s_pt2pt:Pt2pt.state.∀s_mnak:Mnak.state.∀s_bottom:Bottom.state.∀ls,ev,hdr
 RECONFIGURING LAYER Pt2pt::Mnak::Bottom WRAPPED WITH COMPRESSION
    FOR EVENT  DnM(ev, hdr)
    AND STATE  (s_pt2pt, s_mnak, s_bottom)
    ASSUMING   getType ev = ESend  ∧  getPeer ev ≠ ls.rank  ∧  s_bottom.enabled
 YIELDS EVENTS [:DnM(ev,OptSend(Iq.hi s_pt2pt.sends.(getPeer ev), hdr)):]
    AND STATE  ( s_pt2pt[.sends.(getPeer ev) ← Iq.add s_pt2pt.sends.(getPeer ev) (getIov ev) hdr]
               , s_mnak
               , s_bottom
               )
```

5 Code Generation

The fast-track reconfigurations of Ensemble protocol stacks discussed in the previous sections describe how to handle several kinds of common events in a much more efficient way. In order make use of these optimizations in a running application system we have to convert the logical reconfiguration theorems into Ocaml-code that deals with both the common and the less common events. As illustrated in figure 13, we have to introduce a "switch" that identifies common events and uses the reconfigured code to process them while passing the remaining events to the usual protocol stack.

The statements of the reconfiguration theorems describe the *result* of passing certain events through the reconfigured protocol stack, i.e. a queue of events to be

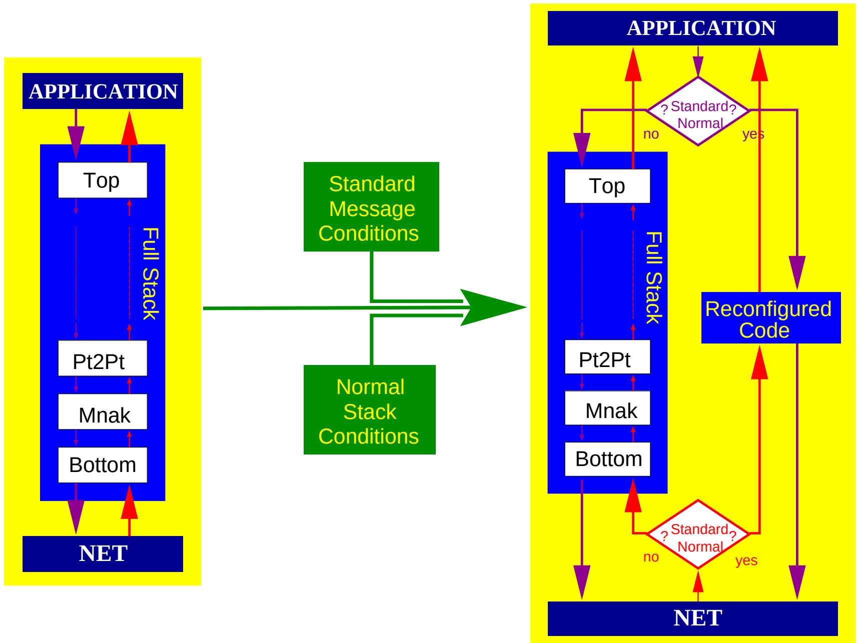

Fig. 13. Stack reconfiguration: original and optimized system

emitted and the modified state of the protocol stack. To transform these results into the code that creates them, we have to convert a substitution on states `s[.field` $\leftarrow$ `expression]` into the assignment `s.field <- expression`. No assignment needs to be generated for layer states which were not modified. A queue of events `[:` $\texttt{ev}_1$ `; ...;` $\texttt{ev}_n$ `:]` has to be converted into a sequence of calls to the event handlers `up` and `dn`, which were introduced by the layer conversion function in figure 3 (page 7). An event of the form `UpM(ev,hdr)` will lead to a call of `up ev hdr` and `DnM(ev,hdr)` will lead to `dn ev hdr`. These conversions can be generated automatically by applying the meta-level function `StatementsOf` to each reconfiguration theorem of a given protocol stack.

To generate the switch we need to look at all the reconfiguration theorems of the protocol stack at once and create a case expression that separates the corresponding pieces of code from each other. Usually, we distinguish four fundamental cases – up- and down-going events as well as sending and broadcasting – which will be expressed by four patterns in the generated code. All assumptions of a reconfiguration theorem that do not deal with the direction or the type of an event will be converted into a conditional, or another case expression if they introduce free variables. As an optimization we also evaluate assumptions about generated events and eliminate assumptions which evaluate to true or are simple variations of other assumptions.

Events that fit one of four patterns and satisfy the conditions of that case will be processed by the corresponding optimized event handler, i.e. the piece of code generated by `StatementsOf`. All other events will be passed to the event handler of the original protocol stack. On the basis of these insights, the meta-level function `ReconfiguredBody` generates an optimized OCAML-implementation of a given protocol stack that follows the conventions of the

```
let opt_stack state (ls,vs) =
   let original_stack = wrap_hdr compress expand
                             (compose Pt2pt.l (compose Mnak.l Bottom.l))
   in
    let up ev hdr (s,q)  = (s, Fqueue.add (UpM(ev,hdr)) q)
    and dn ev hdr (s,q)  = (s, Fqueue.add (DnM(ev,hdr)) q)
    and s,hdlr           = original_stack state (ls,vs)
    in
     let opt_hdlr ((s_pt2pt,s_mnak,s_bottom),event) =
       match event, (getType event) with
       | (DnM(ev, hdr), ESend)
          -> if getPeer ev <> ls.rank  & s_bottom.enabled
               then  s_pt2pt.sends.(getPeer ev)
                      <- Iq.add s_pt2pt.sends.(getPeer ev) (getIov ev) hdr
                   ; dn ev (OptSend(Iq.hi s_pt2pt.sends.(getPeer ev), hdr))
               else hdlr ((s_pt2pt,s_mnak,s_bottom),event)
       | (DnM(ev, hdr), ECast)
          -> if s_bottom.enabled
               then  s_mnak.buf.(ls.rank)
                      <- Iq.opt_insert_doread s_mnak.buf.(ls.rank)
                             (Iq.hi s_mnak.buf.(ls.rank)) ev.iov
                             (Full(NoHdr, hdr))
                   ; dn ev (OptCast(Iq.hi s_mnak.buf.(ls.rank), hdr))
               else hdlr ((s_pt2pt,s_mnak,s_bottom),event)
      | (UpM(ev, OptSend(seqno, hdr)), ESend)
         -> if Iq.opt_update_check s_pt2pt.recvs.(getPeer ev) seqno
                    & not s_bottom.failed.(getPeer ev)
              then  s_pt2pt.recvs.(getPeer ev)
                     <- Iq.opt_update_update s_pt2pt.recvs.(getPeer ev) seqno
                  ; up ev hdr
              else hdlr ((s_pt2pt,s_mnak,s_bottom),event)
      | (UpM(ev, OptCast(seqno, hdr)), ECast)
         -> if Iq.opt_insert_check  s_mnak.buf.(getPeer ev) seqno
                    & not s_bottom.failed.(getPeer ev)
              then s_mnak.buf.(getPeer ev)
                    <- Iq.opt_insert_doread s_mnak.buf.(getPeer ev) seqno
                           ev.iov (Full(NoHdr, hdr))
                   ; up ev hdr
              else hdlr ((s_pt2pt,s_mnak,s_bottom),event)
      | _  ->    hdlr ((s_pt2pt,s_mnak,s_bottom),event)
    in
        (s,opt_hdlr)
```

Fig. 14. Reconfigured code for `Pt2pt::Mnak::Bottom`

functional layer conversion. The result of applying this function to the three-layer stack `Pt2pt::Mnak::Bottom` from example 2 (including the compressions from example 3) is presented in figure 14.

After the code for the optimized stack has been generated as a NuPRL object we prove that the generated code is equivalent to the original one not just for the common events but for all others as well. We then export the code into an Ocaml source file and compile it into executable code. Using the export mechanism discussed in section 1 these steps can be performed fully automatically.

6 The Application Interface

Reliable group communication systems have to deal with many aspects of communication that are not related with the application. Thus the architecture in figure 1, which puts the application on top of the protocol stack, is not the most efficient one, since all application messages have to go through unnecessarily many protocol layers. To overcome this deficiency, Ensemble connects the application to a designated layer `partial_appl` *within* the stack (see figure 15). All

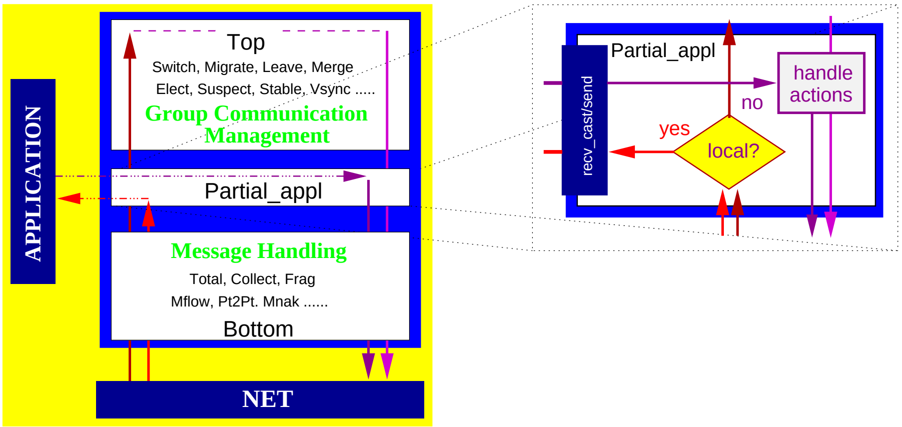

Fig. 15. Application messages leave the stack in the partial_appl layer

layers in the stack below `partial_appl` have to handle data, i.e. messages from the application, and deal with fragmentation, message ordering, point-to-point connections etc. The layers on top of `partial_appl`, however, are only responsible for the management of the communication itself, e.g. with stability, merging, leaving, changing groups, virtual synchronicity, etc. As common messages are usually sent and received by the application, they have nothing to do with these protocols and can leave the stack before entering the respective layers.

This refined architecture makes a fully formal reconfiguration of Ensemble protocol stacks much more complicated, because the interaction between the `partial_appl` layer and the application does not rely on events anymore. Instead, the application has to provide functions `recv_send` and `recv_cast`, which process an incoming message and return a list of actions that are converted into events by `partial_appl`. To identify messages that are directed to the application, the corresponding events receive a header `Local`. Formally, the application has become a part of `partial_appl` and we have reason about the results of the functions `recv_send` and `recv_cast` in order to create a fast-path through the layer `partial_appl`.

A major difference to the reconfiguration of other protocol layers is that we cannot reason independently about up-going events, which eventually will be delivered to the application, and down-going ones, which are initiated by the application. As far as communication is concerned, the events initiated by the application are the results of processing an event that enters the `partial_appl` layer. Thus the clean separation between sending and receiving messages in a fast-track reconfiguration of protocol layers is not possible for `partial_appl`. Furthermore, the number of messages initiated by the application cannot be determined from the code of `partial_appl` and thus reconfiguration theorems as described in section 2.2 cannot be formulated anymore.[6] There are two alternatives for the reconfiguration of protocol stacks with an application interface.

[6] We cannot state assumptions about possible results of applying `recv_cast` and `recv_send` because this would cause them to become a part of the switch between reconfigured and standard event handler. Since the functions have side-effects in the application, they cannot be re-executed if the switch selects the standard handler.

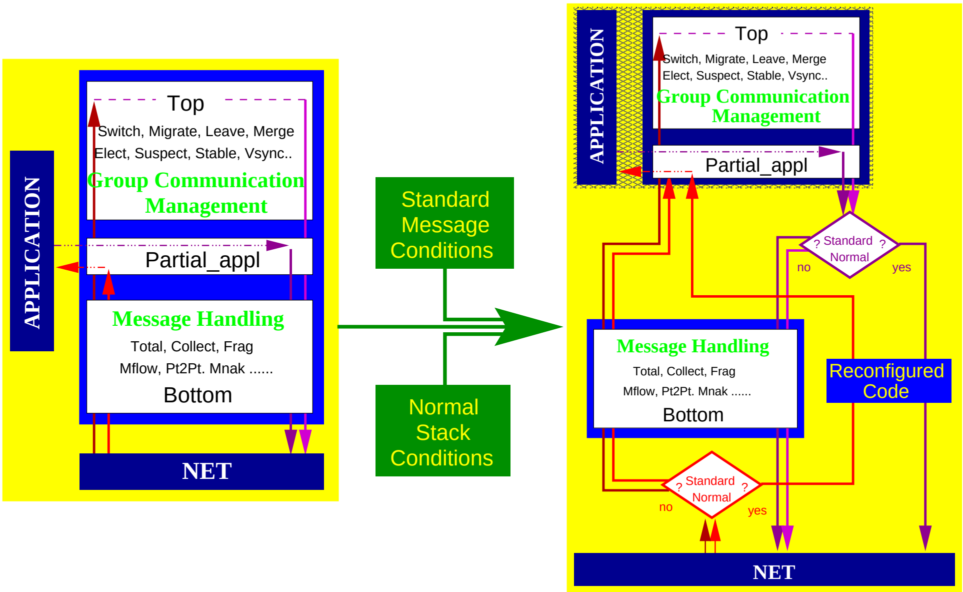

Fig. 16. Stack reconfiguration viewing the upper stack as application

A simple approach, illustrated above, is to consider `partial_appl` and layers above it as part of the application. In this case only the protocol layers below `partial_appl` will be reconfigured while all the other layers remain unchanged. The justification for this approach is that common events do not enter the management stack anyway: a reconfiguration of the whole protocol stack would not affect the management layers above `partial_appl` at all. On the other hand, the stack of layers that deal with message handling can be reconfigured by the tactics discussed so far. In this case, the reconfigured code of the complete protocol stack would have the following structure.

```
let opt_stack =
  let opt_msg_stack state (ls,vs) =
    let msg_stack = wrap_hdr compress expand (compose l_1^mh (compose l_2^mh ... Bottom.l))  in
    let s,hdlr = msg_stack state (ls,vs)    in
    let opt_hdlr
        .
        .
        .
    in
      (s,opt_hdlr)
  in
     compose (compose Top.l (compose l_1^cm (compose l_2^cm ...Partial_appl.l))) opt_msg_stack
```

where the l_i^{cm} denote the layers for communication management and the l_i^{mh} the message handling layers. The advantage of this approach is that it is based entirely on well-understood mechanisms and makes a verification of the resulting optimized system comparably easy, as the protocol stack is clearly separated from the "application". Its disadvantage is a small efficiency loss, because the layer `partial_apply` is not being reconfigured.

The alternative solution is to reconfigure the code of `partial_appl` directly, instead of generating theorems about the result of passing events through the stack, and to compose it with the stack below. For this purpose we have to analyze `partial_appl`'s event handler for local events (see figure 17), to optimize it by interactively applying evaluation rules, and to use specialized tactics for composing the result with the remaining stack. Assuming that the layer is not blocked

```
let  hdlrs s (ls,vs) {up_out=up;upnm_out=upnm;dn_out=dn;dnlm_out=dnlm;dnnm_out=dnnm} =
  let log             = ...
  let check_block_ok  = ...
  let handle_control  = ...
  let handle_actions actions =
     match s.blocked with
      | B -> if Array.length actions <> 0 then failwith "got actions when Blocked"
      | _ -> ()
   ; for i = 0 to pred (Array.length actions) do
       match actions.(i) with
       | Cast(msg)      -> dnlm (castIovAppl name msg) ()
       | Send(dest,msg) ->
          for i = 0 to pred (Array.length dest) do
           let dest = dest.(i) in
             if dest = ls.rank then log (fun () -> sprintf "send to myself dest=%d" dest)
           ; if dest < 0 | dest >= ls.nmembers then failwith "bad destination"
           ; array_incr s.send_xmit dest
           ; dnlm (sendPeerIovAppl name dest msg) ()
          done
       | Control ctl -> handle_control ctl
     done
   in
   let uplm_hdlr ev () = match getType ev with
                         | ECast -> let origin = getPeer ev in
                                    let iov = getIov ev in
                                      handle_actions (s.recv_cast.(origin) iov)
                                    ; ack ev     ; free name ev
                                    ; if s.up_block_ok then check_block_ok ()
                         | ESend -> let origin = getPeer ev in
                                    array_incr s.send_recv origin
                                  ; let iov = getIov ev in
                                      handle_actions (s.recv_send.(origin) iov)
                                    ; ack ev     ; free name ev
                                    ; if s.up_block_ok then check_block_ok ()
                         | _ -> failwith "got bad uplm event"
        :
   in {up_in=up_hdlr;uplm_in=uplm_hdlr;upnm_in=upnm_hdlr;dn_in=dn_hdlr;dnnm_in=dnnm_hdlr}
```

Fig. 17. Event handler of the layer `partial_appl`

(`s_partial_appl.blocked = U`), no acknowledgement is required (`getAck ev = NoAck`), and that there is no up_block (`not s_partial_appl.up_block_ok`) we can remove many redundancies from the code of the event handler for local events and get the following optimized version.

```
let dnlm ev (s,q)  = (s, Fqueue.add (DnM(ev, Local())) q) in
let handle_actions actions =
  for i = 0 to pred (Array.length actions) do
    match actions.(i) with
    | Cast(msg) -> dnlm ({typ=ECast; peer=invalid_rank; iov=msg; extend=[]; applmsg=true})
    | Send(dest,msg) ->
        for i = 0 to pred (Array.length dest) do
          if dest.(i)=ls.rank | dest.(i)<0 | dest.(i)>=ls.nmembers  then fail
        ; s_partial_appl.send_xmit.(dest.(i)) <- succ s_partial_appl.send_xmit.(dest.(i))
        ; dnlm ({typ=ESend; peer=dest.(i); iov=msg; extend=[]; applmsg=true})
        done
    | Control ctl -> handle_control ctl
  done
in
let opt_hdlr (s_partial_appl, ev)  = match getType ev with
    | ECast ->  handle_actions (s_partial_appl.recv_cast.(getPeer ev) ev.iov)
    | ESend ->  s_partial_appl.send_recv.(getPeer ev)
                  <- succ s_partial_appl.send_recv.(getPeer ev)
              ; handle_actions (s_partial_appl.recv_send.(getPeer ev) ev.iov)
    | _     -> failwith "got bad uplm event"
```

Composing this handler with the reconfigured stack below the `partial_appl` layer and the unmodified layers above it leads to the reconfiguration of the application stack described in figure 18. Note that `compress` and `expand` now involve `partial_appl`, which eliminates the variable `hdr` in the definition of `OptSend` and `OptCast`.

```
let opt_stack =
 let opt_app_stack state (ls,vs) =
   let msg_stack = compose l_1^mh (compose l_2^mh ... Bottom.l)
   and app_stack = compose partial_appl msg_stack
   in
   let dn ev hdr (s,q)  = (s, Fqueue.add (DnM(ev,hdr)) q)
   and s_msg,hdlr_msg = wrap_hdr compress expand msg_stack state (ls,vs)
   and s_app,hdlr_app = wrap_hdr compress expand app_stack state (ls,vs)
   in
   let opt_hdlr_app  ((s_partial_appl, s_1^mh, ..., s_bottom),event) =
     let handle_actions actions =
       for i = 0 to pred (Array.length actions) do
         match actions.(i) with
         | Cast(msg)      ->
             let ev = {typ=ECast; peer=invalid_rank; iov=msg; extend=[]; applmsg=true}  in
               if fast-path conditions of msg_stack for cast-events
                  then  cast-state-updates of msg_stack
                        ; dn ev (OptCast(opt-cast-headers of msg_stack))
                  else hdlr_msg ((s_1^mh, ..., s_bottom), DnM(ev, Local()))
         | Send(dest,msg) ->
             for i = 0 to pred (Array.length dest) do
               if dest.(i)=ls.rank | dest.(i)<0 | dest.(i)>=ls.nmembers  then fail
             ; s_partial_appl.send_xmit.(dest.(i)) <- succ s_partial_appl.send_xmit.(dest.(i))
             ; let ev = {typ=ESend; peer=dest.(i); iov=msg; extend=[]; applmsg=true}  in
                 if fast-path conditions of msg_stack for send-events
                   then  send-state-updates of msg_stack
                         ; dn ev (OptSend(opt-send-headers of msg_stack))
                   else hdlr_msg (s_1^mh, ..., s_bottom), DnM(ev, Local()))
             done
         | Control ctl -> handle_control ctl
       done
     in
     match event, (getType event) with
      | (UpM(ev, OptSend( send_vars)), ESend)
         -> if  fast-path conditions of msg_stack for send-events
              & fast-path conditions of partial_appl
            then  send-state-updates of msg_stack
                  ; s_partial_appl.send_recv.(getPeer ev)
                    <- succ s_partial_appl.send_recv.(getPeer ev)
                  ; handle_actions (s_partial_appl.recv_send.(getPeer ev) ev.iov)
            else hdlr_app ((s_partial_appl, s_1^mh, ..., s_bottom), event)
      | (UpM(ev, OptCast( cast_vars), ECast)
         -> if  fast-path conditions of msg_stack for cast-events
              & fast-path conditions of partial_appl
              then  cast-state-updates of msg_stack
                  ; handle_actions (s_partial_appl.recv_cast.(getPeer ev) ev.iov)
              else hdlr_app ((s_partial_appl, s_1^mh, ..., s_bottom), event)
      | _ -> hdlr_app ((s_partial_appl, s_1^mh, ..., s_bottom), event)
   in
       (s_app, opt_hdlr_app)
 in
    compose (compose Top.l (compose l_1^cm (compose l_2^cm...l_n^cm)))  opt_app_stack
```

Fig. 18. Structure of a reconfigured application stack

The advantage of this solution is that the reconfigured protocol stack directly interacts with the application and is very efficient. But, although it is comparably easy to generate the reconfigured code, proving it to be equivalent to the original one might not be so easy, as we can rely less on already verified theorems about the composition of reconfigured layers and have to use tactics to deal with the `partial_appl` layer.

The code given in figure 18 links the fast-path for down-going events to a previous up-going fast-path. Messages to the application that go through the regular protocol stack are handled by the usual event handler, which causes the events initiated by the application to go through the regular stack as well. Only the receiver of these messages may use the reconfigured code to process them. Since almost all messages to the application are to be considered common

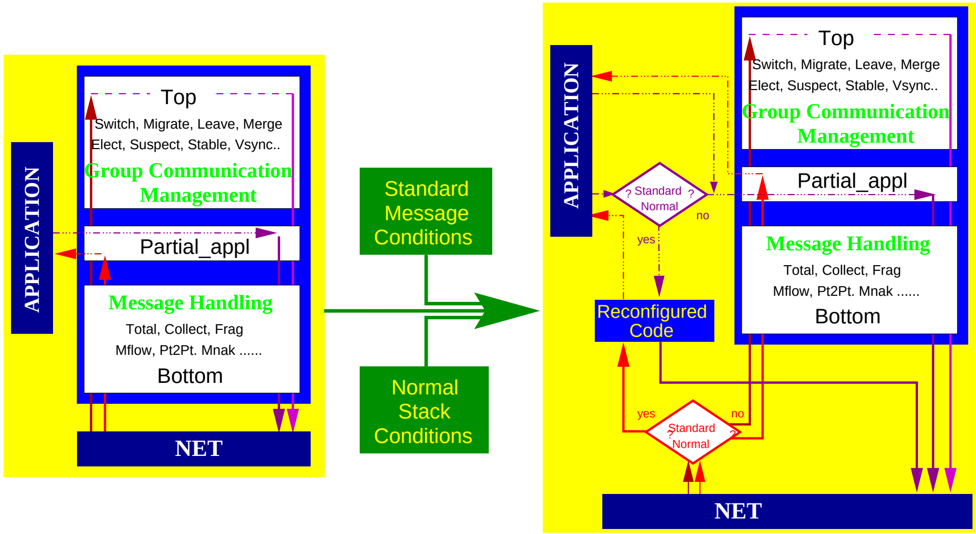

Fig. 19. Stack reconfiguration including the `partial_appl` layer

events, the cases where a down-going event has to go through the regular stack, although the requirements of the fast-path are fulfilled, are rare.[7]

7 Conclusion

We have presented a variety of formal techniques for the reconfiguration of networked application systems built with the ENSEMBLE group communication toolkit. These techniques are implemented as tactics of the NUPRL proof development system and based on an embedding of the actual code of ENSEMBLE into the logical language of NUPRL. The use of NUPRL enabled us to reconfigure the code of ENSEMBLE protocol stacks by combining knowledge about the reconfiguration of individual protocol layers with knowledge about layer composition and header compression. The resulting optimization techniques are completely automated and scale up very well, since they are much more efficient than mechanisms that are entirely based on evaluation. Furthermore, they also prove that the generated code is equivalent to the original one and thus a major aide in the verification of secure and efficient group communication systems.

We are currently applying our reconfiguration techniques to a running application system built with the ENSEMBLE toolkit in order to demonstrate the reliability of our methods and to test the practical performance improvements caused by them. In the future we intend to increase the degree of automation and robustness of our reconfiguration tools and to distribute them with the next release of the ENSEMBLE toolkit. On the other hand we want to extend the reasoning capabilities of our current logical programming environment by techniques for verifying ENSEMBLE's protocol layers [HLvR98] and the application

[7] To deal with this case, one could replace `hdlr_app` by an event handler that contains the reconfigured handler of the stack below `partial_appl` (c.f. figure 16) but this would make the code rather complex.

stacks generated by it. For this purpose we will integrate general deductive tools – such as an extended typechecking algorithm [HK97,HK98], a proof procedure for first-order logic [KOS96], and a proof planner for inductive proofs [PK98] – into the NuPRL system and create tactics that are specifically designed for dealing with the code of Ensemble. By combining all these techniques into a single logical programming environment for Ensemble we hope to create a software development infrastructure for group communication systems that yields the high degree of assurance required by safety-critical applications.

Acknowledgements

The work reported in this paper would not have been possible without the many contributions of Mark Hayden, Jason Hickey and Robbert Van Renessee. The reconfiguration techniques discussed in section 2 are the result of a permanent interaction with Mark Hayden, who also made me able to understand Ensemble architecture and its Ocaml-implementation. Many discussions with Jason Hickey and Robbert Van Renessee helped me to identify fast-paths through the individual protocol layers and to get deeper insights into header compression and the application interface. Regular research meetings with Mark Hayden, Jason Hickey, and Bob Constable led to the conception of the logical programming environment that shall eventually become a part of the Ensemble distribution.

I would also like to thank Bob Constable for initiating and directing this research project and the ongoing encouragement that I have received from him during the past two years.

References

[AH97] M. Archer & C. Heitmeyer. Mechanical verification of timed automata: A case study. Technical report, Naval Research Laboratory, Washington, DC, 1997.

[Bir97] K. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, 1997.

[BvR94] K. Birman & R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[CHTC96] T. Chandra, V. Hadzilacos, S. Toueg, B. Charron-Bost. On the impossibility of group membership. *15th ACM Symposium on Principles of Distributed Computing*, pp. 322–330, 1996.

[CAB+86] R. Constable, et. al., *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, 1986.

[GMW79] M. Gordon, R. Milner, C. Wadsworth. *Edinburgh LCF: A mechanized Logic of Computation.* LNCS 78, Springer Verlag, 1979.

[HK97] O. Hafızoğulları & C. Kreitz. Dead Code Elimination Through Type Inference. Technical Report TR 98-1698, Cornell University, 1998.

[HK98] O. Hafızoğulları & C. Kreitz. A Type-based Framework for Automatic Debugging. Technical Report, Cornell University, 1998.

[Hay96] M. Hayden. *Ensemble Reference Manual.* Cornell University, 1996.

[Hay97] M. Hayden. Distributed communication in ML. Technical Report TR97-1652, Cornell University, 1997.

[Hay98a] The ENSEMBLE distributed communication system. System distribution and documentation. `http://www.cs.cornell.edu/Info/Projects/Ensemble`

[Hay98b] M. Hayden. *The Ensemble System.* PhD thesis, Cornell University, 1998.

[HLvR98] J. Hickey, N. Lynch, R. van Renesse. *Specifications and Proofs for Ensemble Layers.* Technical Report, Cornell University, 1998. (Submitted to TACAS'99)

[HvR96] M. Hayden & R. van Renesse. Optimizing layered communication protocols. Technical Report TR 96-1613, Cornell University, 1996.

[KHH98] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In C. & H. Kirchner, editor, *15^{th} International Conference on Automated Deduction*, LNAI 1421, pp. 317–332, Springer, 1998.

[Kre97] C. Kreitz. Formal reasoning about communication systems I: Embedding ML into type theory. Technical Report TR97-1637, Cornell University, 1997.

[KOS96] C. Kreitz, J. Otten, S. Schmitt. Guiding Program Development Systems by a Connection Based Proof Strategy. In *5th International Workshop on Logic Program Synthesis and Transformation*, LNCS 1048, pp. 137–151. Springer Verlag, 1996.

[Ler97] X. Leroy. *The Objective Caml system release 1.07.* Institut National de Recherche en Informatique et en Automatique, 1998.

[LR93] P. Lincoln & J. Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *23rd Fault-Tolerant Computing Symposium*, pp. 402–411, 1993.

[PK98] B. Pientka & C. Kreitz. Instantiation of existentially quantified variables in inductive specification proofs. In *4^{th} International Conference on Artificial Intelligence and Symbolic Computation*, LNAI, Springer Verlag, 1998.

[Rus92] J. Rushby. Formal methods for dependable real-time systems. In *International Symposium on Real-Time Embedded Processing for Space Applications*, pp. 355–366, 1992.

[Rus97] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. In *Dependable Computing for Critical Applications: 6*, pp. 191–210. IEEE Computer Society, 1997.

[vRBM96] R. van Renesse, K. Birman, & S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

A Functional Implementation of Protocol Layers

The following listings presents the complete ENSEMBLE code for the functional implementation of protocol layers, particularly the definition of events, glueing handlers together, layer installation, layer conversion, composition, and wrapping. The imperative and threaded versions, which we don't list here, can be found in the distribution [Hay98a].

```
(**************************************************************)
(*
 *  Ensemble, (Version 0.50)
 *  Copyright 1998 Cornell University
 *  All rights reserved.
 *
 * EVENT.ML
 * Author: Mark Hayden, 4/95
 * Based on Horus events by Robbert vanRenesse
(**************************************************************)
open Util
open Trans
(**************************************************************)

type acknowledgement =
  | NoAck
  | RankSeqno of origin * seqno

(**************************************************************)

type typ =
    (* These events should have messages associated with them. *)
  | ECast                            (* Multicast message *)
  | ESend                            (* Pt2pt message *)
  | ECastUnrel                       (* Unreliable multicast message *)
  | ESendUnrel                       (* Unreliable pt2pt message *)
  | EMergeRequest                    (* Request a merge *)
  | EMergeGranted                    (* Grant a merge request *)
  | EOrphan                          (* Message was orphaned *)

    (* These types do not have messages. *)
  | EAccount                         (* Output accounting information *)
  | EAck                             (* Acknowledge message *)
  | EAsync                           (* Asynchronous application event *)
  | EBlock                           (* Block the group *)
  | EBlockOk                         (* Acknowledge blocking of group *)
  | EDump                            (* Dump your state (debugging) *)
  | EElect                           (* I am now the coordinator *)
  | EExit                            (* Disable this stack *)
  | EFail                            (* Fail some members *)
  | EGossipExt                       (* Gossip message *)
  | EInit                            (* First event delivered *)
  | EInvalid                         (* Erroneous event type *)
  | ELeave                           (* A member wants to leave *)
  | ELostMessage                     (* Member doesn't have a message *)
  | EMergeDenied                     (* Deny a merge request *)
  | EMergeFailed                     (* Merge request failed *)
  | EMigrate                         (* Change my location *)
  | EPresent                         (* Members present in this view *)
  | EPrompt                          (* Prompt a new view *)
  | EProtocol                        (* Request a protocol switch *)
  | ERekey                           (* Request a rekeying of the group *)
  | EStable                          (* Deliver stability *)
  | EStableReq                       (* Request for stability information *)
  | ESuspect                         (* Member is suspected to be faulty *)
  | ESystemError                     (* Something serious has happened *)
  | ETimer                           (* Request a timer *)
  | EView                            (* Notify that a new view is ready *)
  | EXferDone                        (* Notify that a state transfer is complete *)
  | ESyncInfo

(**************************************************************)
```

```
type field =
      (* Common fields *)
  | Type        of typ                  (* type of the message*)
  | Peer        of rank                 (* rank of sender/destination *)
  | Ack         of acknowledgement      (* acknowledgement information *)
  | Iov         of Iovecl.t             (* payload of message *)

      (* Uncommon fields *)
  | Address     of Addr.set             (* new address for a member *)
  | Failures    of bool Arrayf.t        (* failed members *)
  | Presence    of bool Arrayf.t        (* members present in the current view *)
  | Suspects    of bool Arrayf.t        (* suspected members *)
  | SuspectReason of string             (* reasons for suspicion *)
  | Stability   of seqno Arrayf.t       (* stability vector *)
  | NumCasts    of seqno Arrayf.t       (* number of casts seen *)
  | Mergers     of View.state           (* list of merging members *)
  | Contact     of Endpt.full * View.id option (* contact for a merge *)
  | HealGos     of Proto.id * View.id * Endpt.full * View.t (* HEAL gossip *)
  | SwitchGos   of Proto.id * View.id * Time.t  (* SWITCH gossip *)
  | ExchangeGos of string               (* EXCHANGE gossip *)
  | ViewState   of View.state           (* state of next view *)
  | ProtoId     of Proto.id             (* protocol id *)
  | Time        of Time.t               (* current time *)
  | Alarm       of Time.t               (* for alarm requests *)
  | ApplCasts   of seqno Arrayf.t
  | ApplSends   of seqno Arrayf.t
  | DbgName    of string

      (* Flags *)
  | NoTotal                             (* message is not totally ordered*)
  | ServerOnly                          (* deliver only at servers *)
  | ClientOnly                          (* deliver only at clients *)

      (* Debugging *)
  | ApplMsg                             (* was this message generated by an appl? *)
  | History     of string               (* debugging history *)

      (* A hack... *)
  | Forcecopy

(**************************************************************)
type t = {
  typ     : typ ;
  peer    : rank ;
  iov     : Iovecl.t ;
  applmsg : bool ;
  extend  : field list
}

(**************************************************************)
type dir = Up of t | Dn of t

type ('a,'b) dirm = UpM of t * 'a | DnM of t * 'b
  .
  .
(**************************************************************)
let getExtender f g =
  let rec loop l =
    match l with
    | [] -> g
    | h::t ->
        match f h with
        | Some o -> o
        | None -> loop t
  in fun e -> loop e.extend

let getExtend e   = e.extend
let getIov e      = e.iov
let getPeer e     = e.peer
let getType e     = e.typ
let getApplMsg e = e.applmsg
let getAck        = getExtender (function (Ack i) -> Some i | _ -> None) NoAck
  .
  .
```

```
(**************************************************************)
(*
 *  Ensemble, (Version 0.50)
 *  Copyright 1998 Cornell University
 *  All rights reserved.
 *
 * LAYER.ML
 * Author: Mark Hayden, 4/95
(**************************************************************)
open Util
(**************************************************************)

type ('a,'b,'c) handlers_out = {
  up_out     : Event.up -> 'c -> unit ;
  upnm_out   : Event.up -> unit ;
  dn_out     : Event.dn -> 'c -> 'b -> unit ;
  dnlm_out   : Event.dn -> 'a -> unit ;
  dnnm_out   : Event.dn -> unit
}

type ('a,'b,'c) handlers_in = {
  up_in      : Event.up -> 'c -> 'b -> unit ;
  uplm_in    : Event.up -> 'a -> unit ;
  upnm_in    : Event.up -> unit ;
  dn_in      : Event.dn -> 'c -> unit ;
  dnnm_in    : Event.dn -> unit
}
(* What messages look like in this architecture.  *)
type ('local,'hdr,'abv) msg =
  (* These come first in order to enable some hacks.  *)
  | Local_nohdr
  | Local_seqno of Trans.seqno

  | NoMsg
  | Local of 'local
  | Full of 'hdr * 'abv
  | Full_nohdr of 'abv
  .
  .
  .
(**************************************************************)

(* Layer conversion functions. *)

let hdr init hdlrs args vs =
  let state = init args vs in
  let l {up_lout=up;dn_lout=dn} =
    let up ev abv        = up ev abv
    and upnm ev          = up ev NoMsg
    and dnnm ev          = dn ev NoMsg
    and dn ev abv hdr    = dn ev (Full(hdr,abv))
    and dnlm ev hdr      = dn ev (Local hdr)
    in

    let {up_in=up;uplm_in=uplm;upnm_in=upnm;dn_in=dn;dnnm_in=dnnm} =
      hdlrs state vs {up_out=up;upnm_out=upnm;dn_out=dn;dnlm_out=dnlm;dnnm_out=dnnm}
    in

    let up ev msg = match msg with
    | Local hdr      -> uplm ev      hdr
    | Full(hdr,abv) -> up   ev abv hdr
    | NoMsg          -> upnm ev
    | _              -> failwith "hdr_noopt:sanity"
    and dn ev abv = match abv with
    | NoMsg          -> dnnm ev
    | _              -> dn ev abv
    in

    {up_lin=up;dn_lin=dn}
  in (state,l)
(**************************************************************)
  .
  .
  .
```

```
(* Layer installation. *)

type ('bel,'abv) handlers_lout = {
  up_lout    : Event.up -> 'bel -> unit ;
  dn_lout    : Event.dn -> 'abv -> unit
}

type ('bel,'abv) handlers_lin  = {
  up_lin     : Event.up -> 'abv -> unit ;
  dn_lin     : Event.dn -> 'bel -> unit
}

type ('bel,'abv,'state) basic =
  state -> View.full -> 'state * (('bel,'abv) handlers_lout -> ('bel,'abv) handlers_lin)

type lyr = (unit,unit,unit) basic

let table =
  let table = Hashtbl.create name 199 in
  Trace.install_root (fun () ->
    [sprintf "LAYER:#layers=%d" (Hashtbl.size table)]
  ) ;
  table

let install name layer =
  let layer = (layer : ('a,'b,'c) basic) in
  let layer = (Obj.magic layer : lyr) in
  let name = string_uppercase name in
  Hashtbl.add table name layer

let lookup name =
  let name = string_uppercase name in
  Hashtbl.find table name
```

```
(**************************************************************)
(*
 *  Ensemble, (Version 0.50)
 *  Copyright 1998 Cornell University
 *  All rights reserved.
 *
 * GLUE.ML
 * Author: Mark Hayden, 10/96
(**************************************************************)
open Util
open Event
open Layer
open View
(**************************************************************)
let name = "GLUE"
let failwith s = failwith (Util.failmsg name s)
(**************************************************************)

(* Called to inject the EInit event into a stack.
 *)
let inject_init debug sched up bot_nomsg =
  (* MH:disabled the sched_enqueue because of race condition
   * with messages coming off of the network.
   *)
  up (create "init_protocol" EInit[]) bot_nomsg ;

(**************************************************************)

module type S =
  sig
    type ('state,'a,'b) t
    val convert : ('a,'b,'state) Layer.basic -> ('state,'a,'b) t
    val revert  : ('state,'a,'b) t -> ('a,'b,'state) Layer.basic
    val wrap_hdr : ('a -> 'b) -> ('b -> 'a) -> ('c, 'd, 'a) t -> ('c, 'd, 'b) t
    val compose : ('s1,'a,'b) t -> ('s2,'b,'c) t -> ('s1*'s2,'a,'c) t
    type ('state,'top,'bot) init =
      ('state,'top,'bot) t ->
      'top -> 'bot ->
      Sched.t ->
      Layer.state ->
      View.full ->
      (Event.up -> unit) ->
      (Event.dn -> unit)
    val init : ('state,'top,('a,'b,'c)Layer.msg) init
  end

(**************************************************************)

module Functional : S with
  type ('state,'b,'a) t =
    Layer.state ->
    View.full ->
    'state * (('state * ('a,'b) Event.dirm) ->
              ('state * ('b,'a) Event.dirm Fqueue.t))
= struct
  let name = "Functional"
  let failwith s = failwith (Util.failmsg name s)

  type ('state,'b,'a) t =
    Layer.state ->
    View.full ->
    'state * (('state * ('a,'b) Event.dirm) ->
              ('state * ('b,'a) Event.dirm Fqueue.t))

  type ('state,'top,'bot) init =
    ('state,'top,'bot) t ->
    'top -> 'bot ->
    Sched.t ->
    Layer.state ->
    View.full ->
    (Event.up -> unit) ->
    (Event.dn -> unit)
```

```
 let convert l args view =
   let up ev hdr (s,q)  = (s, Fqueue.add (UpM(ev,hdr)) q)
   and dn ev hdr (s,q)  = (s, Fqueue.add (DnM(ev,hdr)) q)
   and s,h              = l args view                      in
   let {up_lin=up;dn_lin=dn} = h {up_lout=up;dn_lout=dn} in
   let hdlr (s,ev) = match ev with
       | UpM(ev,hdr) -> up ev hdr (s, Fqueue.empty)
       | DnM(ev,hdr) -> dn ev hdr (s, Fqueue.empty)
   in
       (s,hdlr)

let revert f =
  let b state vf =
    let s,h = f state vf in
    let h {up_lout=up;dn_lout=dn} =
      let emit fq =
        Fqueue.iter (function
          | UpM(ev,hdr) ->
              up ev hdr
          | DnM(ev,hdr) ->
              dn ev hdr
        ) fq
      in
      let up ev hdr =
        let (_,evs) = h (s,(UpM(ev,hdr))) in
        emit evs
      and dn ev hdr =
        let (_,evs) = h (s,(DnM(ev,hdr))) in
        emit evs
      in
      {up_lin=up;dn_lin=dn}
    in
    (s,h)
  in b

let wrap_hdr compress expand l state vs =
  let (s,h) = l state vs in
  let emit (s,e) =
    let e =
      Fqueue.map (function
        | UpM(ev,hdr) -> UpM(ev,hdr)
        | DnM(ev,hdr) ->
            let hdr = compress hdr in
            DnM(ev,hdr)
      ) e
    in
    (s,e)
  in

  let h (s,ev) =
    match ev with
    | UpM(ev,hdr) ->
        let hdr = expand hdr in
        emit (h (s,(UpM(ev,hdr))))
    | DnM(ev,hdr) ->
        emit (h (s,(DnM(ev,hdr))))
  in
  (s,h)
```

```
(************************************************************************
   Layer composition: compose 2 converted layers top and bot.
   Make sure that events passed between these two layers are not
   emitted but handled by the respective layers. Events emitted
   by either of the two layers are therefore collected in two event
   lists (queues).

      emit: events to be emitted
      midl: events passed between the layers

   The functions split_top, split_bot will send outgoing Up and Dn
   events into the appropriate list.

   The combined handler has to operate on a pair of states and
   a single event. The event will be passed through the appropriate
   layer and the emitted events will be split accordingly.

   The functions Fqueue.fold and Fqueue.loop are recursive functions
   that operate on a function f: 'a * 'b -> a, an argument i : a, and
   queue q : a (Fqueue.t)
 ************************************************************************)

(* SPLIT: (up,dn) q adds the up and dn events to the up and dn queues   *
 *                  and returns the resulting queues.                    *)
  let split updn b =
    Fqueue.fold (fun (up,dn) ev ->
      match ev with
      | UpM(ev,hdr) -> let up = Fqueue.add (UpM(ev,hdr)) up in  (up,dn)
      | DnM(ev,hdr) -> let dn = Fqueue.add (DnM(ev,hdr)) dn in  (up,dn)
    ) updn b

(* SPLIT_TOP: splits the output events of the upper layer *)
  let split_top emit_midl q = split emit_midl q

(* SPLIT_BOT: Same as split_top except this is for the bottom layer, so  *
             emit and midl get reversed before/after the call to SPLIT. *)
  let split_bot (emit,midl) q =
    let midl,emit = split (midl,emit) q in   (emit,midl)

(* A totally functional layer composition function   *)
  let compose top bot state vf =
    let s1,top = top state vf in      (* Initialize the two layers *)
    let s2,bot = bot state vf in
    let loop (s1,s2) (emit,midl) =
      Fqueue.loop (fun ((s1,s2),emit) midl ev ->
        match ev with
        | UpM(ev,hdr) -> let s1,evs = top (s1,UpM(ev,hdr)) in
                         let (emit,midl) = split_top (emit,midl) evs in
                          (((s1,s2),emit),midl)

        | DnM(ev,hdr) -> let s2,evs = bot (s2,DnM(ev,hdr)) in
                         let (emit,midl) = split_bot (emit,midl) evs in
                          (((s1,s2),emit),midl)

      ) ((s1,s2),emit) midl
    in
                  (* Outer handler takes a single event and then passes it to *
                   * appropriate layer and then splits the emitted events.    *)
    let hdlr = function
      | ((s1,s2),DnM(ev,hdr)) ->
           let s1,emitted = top (s1,DnM(ev,hdr)) in
            let (emit,midl) = split_top (Fqueue.empty,Fqueue.empty) emitted in
             loop (s1,s2) (emit,midl)
      | ((s1,s2),UpM(ev,hdr)) ->
           let s2,emitted = bot (s2,UpM(ev,hdr)) in
            let (emit,midl) = split_bot (Fqueue.empty,Fqueue.empty) emitted in
             loop (s1,s2) (emit,midl)
    in
       ((s1,s2),hdlr)
```

```
  let init l top_msg bot_nomsg sched state vf up_out =
    let dn_r = ref (fun _ _ -> failwith "premature event") in
    let dn ev hdr = !dn_r ev hdr in
    let up ev hdr = up_out ev in
    let s,hdlr = l state vf in
    let up ev hdr =
      let emit = snd (hdlr (s,(UpM(ev,hdr)))) in
      Fqueue.iter (function
        | UpM(ev,hdr) -> up ev hdr
        | DnM(ev,hdr) -> dn ev hdr
      ) emit
    and dn ev =
      let emit = snd (hdlr (s,(DnM(ev,top_msg)))) in
      Fqueue.iter (function
        | UpM(ev,hdr) -> up ev hdr
        | DnM(ev,hdr) -> dn ev hdr
      ) emit
    in
    dn_r := Config_trans.f bot_nomsg vf up ;
    inject_init name sched up bot_nomsg ;
    dn
end
(**************************************************************)
module Imperative : S = struct
  .
  .
module Threader : S = struct
  .
  .
(**************************************************************)
type glue =
  | Imperative
  | Functional
  | Threaded

type ('s,'a,'b) t =
  | Imp of ('s,'a,'b) Imperative.t
  | Fun of ('s,'a,'b) Functional.t
  | Thr of ('s,'a,'b) Threader.t

type ('s,'top,'bot) init = ('s,'top,'bot) t ->
             'top -> 'bot ->
               Sched.t ->
                 Layer.state ->
                   View.full ->
                     (Event.up -> unit) ->
                        (Event.dn -> unit)

let convert g l =
  match g with
  | Imperative -> Imp (Imperative.convert l)
  | Functional -> Fun (Functional.convert l)
  | Threaded   -> Thr (Threader  .convert l)

let revert = function
| Imp(l) -> Imperative.revert l
| Fun(l) -> Functional.revert l
| Thr(l) -> Threader  .revert l

let wrap_hdr compress expand = function
| Imp(l) -> Imp(Imperative.wrap_hdr compress expand l)
| Fun(l) -> Fun(Functional.wrap_hdr compress expand l)
| Thr(l) -> Thr(Threader  .wrap_hdr compress expand l)

let compose l1 l2 = match l1,l2 with
| Imp(l1),Imp(l2) -> Imp(Imperative.compose l1 l2)
| Fun(l1),Fun(l2) -> Fun(Functional.compose l1 l2)
| Thr(l1),Thr(l2) -> Thr(Threader  .compose l1 l2)
| _,_ -> failwith "mismatched layers"

let init = function
| Imp(l) -> Imperative.init l
| Fun(l) -> Functional.init l
| Thr(l) -> Threader  .init l
(**************************************************************)
```

B Library Listing of Reconfigure.prl

In this section we list the most important abstractions and theorems of our reconfiguration library. Besides formal notations it consists of reconfiguration theorems for the individual protocol layers and the theorems about layer composition and compression. All Parameters that occur on the right hand side of an abstraction but not on its left hand side are a part of the newly defined NuPRL term but suppressed from the display.

```
*******************************************************************************************
***                                Theory RECONFIGURE                                   ***
*******************************************************************************************

* ABS Hidden_all          B[x]  ==   ∀x:A. B[x]

* ABS RECONF               RECONFIGURE LAYER layer
                             FOR EVENT event[ev; hdr]
                             AND STATE s_layer
                           ASSUMING  assumptions[ls; ev]
                         ==
                           ∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
                            ∀ev:Event.t.∀hdr:Headers.∀args:Arguments.
                               assumptions[ls; ev]
                              ⇒ let Fun(l)  = layer in
                                 let s_init, hdlr  = l args (ls, vs) in
                                   hdlr ( s_layer,  event[ev; hdr])

* ABS RECONF_RESULT       ∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
                           RECONFIGURING LAYER  layer
                              FOR EVENT   event[ev; hdr]
                              AND STATE   s_layer
                              ASSUMING    assumptions[ls; ev]
                           YIELDS EVENTS event_queue[ls; ev; hdr]
                              AND STATE   state[ls; ev; hdr]
                         ==
                           ∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
                            ∀ev:Event.t.∀hdr:Headers.∀args:Arguments.
                               assumptions[ls; ev]
                              ⇒ let Fun(l)  =  layer in
                                 let s_init, hdlr  = l args (ls, vs) in
                                  hdlr ( s_layer,  event[ev; hdr])
                                = ( state[ls; ev; hdr],  event_queue[ls; ev; hdr])

* ABS RECONF_EQ           RECONFIGURING LAYER layer
                              FOR EVENT   event
                              AND STATE   s_layer
                           YIELDS EVENTS event_queue
                              AND STATE   state
                         ==
                           let Fun(l)  = layer in
                            let s_init, hdlr  = l args (ls, vs) in
                             hdlr (s_layer, event)
                           = (state, event_queue)

* ABS CONVERT              $module  ==  convert Functional $module.layer

* ABS COMPOSE              Upper::Lower  ==  compose Upper Lower

* ABS WRAP_COMPRESS       layer  WRAPPED WITH COMPRESSION
                         ==
                           wrap_hdr compress expand layer

* ABS HIDE                 ..... Code hidden ....   ==  t

* ABS Fqueue.expression    [:ev; evs:]  ==  Fqueue.add ev evs
```

```
**************************************************************************
***               Layer Reconfiguration Theorems                      ***
**************************************************************************

* THM BottomReconfUpMESend_verif
  ∀s_bottom:Bottom.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Bottom
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_bottom
       ASSUMING   getType ev = ESend   ∧ not ((Arrayf.get s_bottom.failed (getPeer ev)))
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_bottom

* THM BottomReconfUpMECast_verif
  ∀s_bottom:Bottom.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Bottom
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_bottom
       ASSUMING   getType ev = ECast   ∧ not ((Arrayf.get s_bottom.failed (getPeer ev)))
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_bottom

* THM BottomReconfDnMESend_verif
  ∀s_bottom:Bottom.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Bottom
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_bottom
       ASSUMING   getType ev = ESend   ∧ s_bottom.enabled
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_bottom

* THM BottomReconfDnMECast_verif
  ∀s_bottom:Bottom.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Bottom
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_bottom
       ASSUMING   getType ev = ECast   ∧ s_bottom.enabled
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_bottom

****************************************

* THM MnakReconfUpMESend_verif
  ∀s_mnak:Mnak.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Mnak
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_mnak
       ASSUMING   getType ev = ESend
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_mnak

* THM MnakReconfUpMECast_verif
  ∀s_mnak:Mnak.state.∀seqno:Trans.seqno.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Mnak
       FOR EVENT  UpM(ev, Full(Data(seqno), hdr))
       AND STATE  s_mnak
       ASSUMING   getType ev = ECast
               ∧  (Iq.opt_insert_check (Arraye.get s_mnak.buf (getPeer ev)) seqno)
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_mnak[.buf.(getPeer ev)
                        ← Iq.opt_insert_doread s_mnak.buf.(getPeer ev) seqno (getIov ev) hdr]

* THM MnakReconfDnMESend_verif
  ∀s_mnak:Mnak.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Mnak
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_mnak
       ASSUMING   getType ev = ESend
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_mnak

* THM MnakReconfDnMECast_verif
  ∀s_mnak:Mnak.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Mnak
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_mnak
       ASSUMING   getType ev = ECast
    YIELDS EVENTS [:DnM(ev, Full(Data(Iq.hi s_mnak.buf.(ls.rank)), hdr)):]
       AND STATE  s_mnak[.buf.(ls.rank)
                        ← Iq.opt_insert_doread s_mnak.buf.(ls.rank)
                                   (Iq.hi s_mnak.buf.(ls.rank)) ev.iov hdr]
```

```
* THM Pt2ptReconfUpMESend_verif
  ∀s_pt2pt:Pt2pt.state.∀seqno:Trans.seqno.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Pt2pt
       FOR EVENT  UpM(ev, Full(Data(seqno), hdr))
       AND STATE  s_pt2pt
       ASSUMING   getType ev = ESend
                ∧ (Iq.opt_update_check (Arraye.get s_pt2pt.recvs (getPeer ev)) seqno)
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_pt2pt[.recvs.(getPeer ev)
                           ← Iq.opt_update_update s_pt2pt.recvs.(getPeer ev) seqno]

* THM Pt2ptReconfUpMECast_verif
  ∀s_pt2pt:Pt2pt.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Pt2pt
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_pt2pt
       ASSUMING   getType ev = ECast
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_pt2pt

* THM Pt2ptReconfDnMESend_verif
  ∀s_pt2pt:Pt2pt.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Pt2pt
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_pt2pt
       ASSUMING   getType ev = ESend  ∧ not (getPeer ev = ls.rank)
    YIELDS EVENTS [:DnM(ev, Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)), hdr)):]
       AND STATE  s_pt2pt[.sends.(getPeer ev)
                           ← Iq.add s_pt2pt.sends.(getPeer ev) (getIov ev) hdr]

* THM Pt2ptReconfDnMECast_verif
  ∀s_pt2pt:Pt2pt.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Pt2pt
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_pt2pt
       ASSUMING   getType ev = ECast
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_pt2pt

****************************************

* THM MflowReconfUpMESend_verif
  ∀s_mflow:Mflow.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Mflow
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_mflow
       ASSUMING   getType ev = ESend
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_mflow

* THM MflowReconfUpMECast_verif
  ∀s_mflow:Mflow.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Mflow
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_mflow
       ASSUMING   getType ev = ECast
                ∧ getPeer ev <> ls.rank
                ∧ not (Mcredit.got_msg s_mflow.credit (getPeer ev)(msg_len s_mflow ev)
                       >= s_mflow.ack_thresh)
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_mflow

* THM MflowReconfDnMESend_verif
  ∀s_mflow:Mflow.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Mflow
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_mflow
       ASSUMING   getType ev = ESend
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_mflow

* THM MflowReconfDnMECast_verif
  ∀s_mflow:Mflow.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Mflow
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_mflow
       ASSUMING   getType ev = ECast  ∧ (getApplMsg ev)  ∧ (Mcredit.check s_mflow.credit)
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_mflow[.credit ← Mcredit.take s_mflow.credit (msg_len s_mflow ev)]
```

```
* THM Pt2ptwReconfUpMESend_verif
  ∀s_pt2ptw:Pt2ptw.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Pt2ptw
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_pt2ptw
       ASSUMING   getType ev = ESend
                ∧ not (s_pt2ptw.recv_credit.(getPeer ev) > s_pt2ptw.ack_thresh)
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_pt2ptw[.recv_credit.(getPeer ev)
                     ← s_pt2ptw.recv_credit.(getPeer ev)+msg_len s_pt2ptw ev]

* THM Pt2ptwReconfUpMECast_verif
  ∀s_pt2ptw:Pt2ptw.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Pt2ptw
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_pt2ptw
       ASSUMING   getType ev = ECast
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_pt2ptw

* THM Pt2ptwReconfDnMESend_verif
  ∀s_pt2ptw:Pt2ptw.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Pt2ptw
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_pt2ptw
       ASSUMING   getType ev = ESend
                ∧ not ((Arrayf.get s_pt2ptw.failed (getPeer ev)))
                ∧ s_pt2ptw.send_credit.(getPeer ev) > 0
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_pt2ptw[.send_credit.(getPeer ev)
                     ← s_pt2ptw.send_credit.(getPeer ev) - msg_len s_pt2ptw ev]

* THM Pt2ptwReconfDnMECast_verif
  ∀s_pt2ptw:Pt2ptw.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Pt2ptw
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_pt2ptw
       ASSUMING   getType ev = ECast
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_pt2ptw

****************************************

* THM FragReconfUpMESend_verif
  ∀s_frag:Frag.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Frag
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_frag
       ASSUMING   getType ev = ESend
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_frag

* THM FragReconfUpMECast_verif
  ∀s_frag:Frag.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Frag
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_frag
       ASSUMING   getType ev = ECast
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_frag

* THM FragReconfDnMESend_verif
  ∀s_frag:Frag.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Frag
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_frag
       ASSUMING   getType ev = ESend   ∧ Iovecl.len name (getIov ev) <= s_frag.max_len
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_frag

* THM FragReconfDnMECast_verif
  ∀s_frag:Frag.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Frag
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_frag
       ASSUMING   getType ev = ECast   ∧ Iovecl.len name (getIov ev) <= s_frag.max_len
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_frag
```

```
* THM LocalReconfUpMESend_verif
  ∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Local
       FOR EVENT  UpM(ev, Full((), hdr))
       AND STATE  ()
       ASSUMING   getType ev = ESend
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  ()

* THM LocalReconfUpMECast_verif
  ∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Local
       FOR EVENT  UpM(ev, Full((), hdr))
       AND STATE  ()
       ASSUMING   getType ev = ECast
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  ()

* THM LocalReconfDnMESend_verif
  ∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Local
       FOR EVENT  DnM(ev, hdr)
       AND STATE  ()
       ASSUMING   getType ev = ESend   ∧ not (ls.rank = getPeer ev)
    YIELDS EVENTS [:DnM(ev, Full((), hdr)):]
       AND STATE  ()

* THM LocalReconfDnMECast_verif
  ∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Local
       FOR EVENT  DnM(ev, hdr)
       AND STATE  ()
       ASSUMING   getType ev = ECast
    YIELDS EVENTS [:UpM( peer=ls.rank;  extend=[];  typ=ECast;
                         iov=getIov ev;  applmsg = false , hdr)
                   ; DnM(ev, Full((), hdr)) :]
       AND STATE  ()

****************************************

* THM CollectReconfUpMESend_verif
  ∀s_collect:Collect.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Collect
       FOR EVENT  UpM(ev, Full((), hdr))
       AND STATE  s_collect
       ASSUMING   getType ev = ESend
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_collect

* THM CollectReconfUpMECast_verif
  ∀s_collect:Collect.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Collect
       FOR EVENT  UpM(ev, Full((), hdr))
       AND STATE  s_collect
       ASSUMING   getType ev = ECast   ∧ not (s_collect.up_block_ok)
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_collect[.cast_recv.(getPeer ev)
                             ← succ s_collect.cast_recv.(getPeer ev)]

* THM CollectReconfDnMESend_verif
  ∀s_collect:Collect.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Collect
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_collect
       ASSUMING   getType ev = ESend
    YIELDS EVENTS [:DnM(ev, Full((), hdr)):]
       AND STATE  s_collect

* THM CollectReconfDnMECast_verif
  ∀s_collect:Collect.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Collect
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_collect
       ASSUMING   getType ev = ECast
    YIELDS EVENTS [:DnM(ev, Full((), hdr)):]
       AND STATE  s_collect[.cast_xmit ← succ s_collect.cast_xmit]
```

```
* THM TotalReconfUpMESend_verif
  ∀s_total:Total.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Total
       FOR EVENT  UpM(ev, Full(NoHdr, hdr))
       AND STATE  s_total
       ASSUMING   getType ev = ESend   ∧  (getAck ev) = NoAck
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_total

* THM TotalReconfUpMECast_verif
  ∀s_total:Total.state.∀token:Total.token.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Total
       FOR EVENT  UpM(ev, Full(Ordered(token), hdr))
       AND STATE  s_total
       ASSUMING   getType ev = ECast
                ∧ (Iq.opt_update_check s_total.order token)
                ∧ getAck ev = NoAck
    YIELDS EVENTS [:UpM(ev, hdr):]
       AND STATE  s_total[.order ← Iq.opt_update_update s_total.order token]

* THM TotalReconfDnMESend_verif
  ∀s_total:Total.state.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Total
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_total
       ASSUMING   getType ev = ESend   ∧ not (s_total.acct)
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, hdr)):]
       AND STATE  s_total

* THM TotalReconfDnMECast_verif
  ∀s_total:Total.state.∀token:Total.token.∀ls:View.local.∀ev:Event.t.∀hdr:Headers.
    RECONFIGURING LAYER Total
       FOR EVENT  DnM(ev, hdr)
       AND STATE  s_total
       ASSUMING   getType ev = ECast   ∧ s_total.token = Some(token)   ∧ not (s_total.acct)
    YIELDS EVENTS [:DnM(ev, Full(Ordered(token), hdr)):]
       AND STATE  s_total[.token ← Some(succ token)]
```

```
****************************************************************************
***                      Composition Theorems                            ***
****************************************************************************

* THM ComposeUpLinear
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower:State_lower.
  ∀s_upper,s1_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev:Event.t.∀hdr,hdr1,hdr2:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧  Upper = Fun(upper)
    ⇒ RECONFIGURING LAYER Lower
          FOR EVENT  UpM(ev, hdr)          AND STATE  s_lower
       YIELDS EVENTS [:UpM(ev, hdr1):]     AND STATE  s1_lower

       ∧ RECONFIGURING LAYER Upper
            FOR EVENT  UpM(ev, hdr1)       AND STATE  s_upper
         YIELDS EVENTS [:UpM(ev, hdr2):]   AND STATE  s1_upper

    ⇒ RECONFIGURING LAYER  Upper::Lower
          FOR EVENT  UpM(ev, hdr)          AND STATE  (s_upper, s_lower)
       YIELDS EVENTS [:UpM(ev, hdr2):]     AND STATE  (s1_upper, s1_lower)

* THM ComposeUpBounce
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower,s2_lower:State_lower.
  ∀s_upper,s1_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧  Upper = Fun(upper)
    ⇒ RECONFIGURING LAYER Lower
          FOR EVENT  UpM(ev, hdr)          AND STATE  s_lower
       YIELDS EVENTS [:UpM(ev, hdr1):]     AND STATE  s1_lower

       ∧ RECONFIGURING LAYER Upper
            FOR EVENT  UpM(ev, hdr1)       AND STATE  s_upper
         YIELDS EVENTS [:DnM(ev1, hdr2):]  AND STATE  s1_upper

       ∧ RECONFIGURING LAYER Lower
            FOR EVENT  DnM(ev1, hdr2)      AND STATE  s1_lower
         YIELDS EVENTS [:DnM(ev1, hdr3):]  AND STATE  s2_lower

    ⇒ RECONFIGURING LAYER  Upper::Lower
          FOR EVENT  UpM(ev, hdr)          AND STATE  (s_upper, s_lower)
       YIELDS EVENTS [:DnM((ev1, hdr3)):]  AND STATE  (s1_upper, s2_lower)

* THM ComposeUpSplit
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower,s2_lower:State_lower.
  ∀s_upper,s1_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3,hdr4:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧  Upper = Fun(upper)
    ⇒ RECONFIGURING LAYER Lower
          FOR EVENT  UpM(ev, hdr)          AND STATE  s_lower
       YIELDS EVENTS [:UpM(ev, hdr1):]     AND STATE  s1_lower

       ∧ RECONFIGURING LAYER Upper
            FOR EVENT  UpM(ev, hdr1)       AND STATE  s_upper
         YIELDS EVENTS [:UpM(ev, hdr2); DnM(ev1, hdr3):]
            AND STATE  s1_upper

       ∧ RECONFIGURING LAYER Lower
            FOR EVENT  DnM(ev1, hdr3)      AND STATE  s1_lower
         YIELDS EVENTS [:DnM(ev1, hdr4):]  AND STATE  s2_lower

    ⇒ RECONFIGURING LAYER Upper::Lower
          FOR EVENT  UpM(ev, hdr)          AND STATE  (s_upper, s_lower)
       YIELDS EVENTS [:DnM(ev1, hdr4); UpM(ev, hdr2):]
          AND STATE  (s1_upper, s2_lower)
```

```
* THM ComposeUpSplit1
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower,s2_lower:State_lower.
  ∀s_upper,s1_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3,hdr4:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧ Upper = Fun(upper)
   ⇒ RECONFIGURING LAYER Lower
        FOR EVENT  UpM(ev, hdr)          AND STATE  s_lower
     YIELDS EVENTS [:UpM(ev, hdr1):]     AND STATE  s1_lower

      ∧ RECONFIGURING LAYER Upper
          FOR EVENT  UpM(ev, hdr1)       AND STATE  s_upper
       YIELDS EVENTS [:DnM(ev1, hdr2); UpM(ev, hdr3):]
          AND STATE  s1_upper

      ∧ RECONFIGURING LAYER Lower
          FOR EVENT  DnM(ev1, hdr2)      AND STATE  s1_lower
       YIELDS EVENTS [:DnM(ev1, hdr4):] AND STATE  s2_lower

   ⇒ RECONFIGURING LAYER Upper::Lower
        FOR EVENT  UpM(ev, hdr)          AND STATE  (s_upper, s_lower)
     YIELDS EVENTS [:DnM(ev1, hdr4); UpM(ev, hdr3):]
        AND STATE  (s1_upper, s2_lower)

* THM ComposeUpSplit2
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower:State_lower.
  ∀s_upper,s1_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧ Upper = Fun(upper)
   ⇒ RECONFIGURING LAYER Lower
        FOR EVENT  UpM(ev, hdr)          AND STATE  s_lower
     YIELDS EVENTS [:UpM(ev, hdr1); DnM(ev1, hdr2):]
        AND STATE  s1_lower

      ∧ RECONFIGURING LAYER Upper
          FOR EVENT  UpM(ev, hdr1)       AND STATE  s_upper
       YIELDS EVENTS [:UpM(ev, hdr3):]   AND STATE  s1_upper

   ⇒ RECONFIGURING LAYER Upper::Lower
        FOR EVENT  UpM(ev, hdr)          AND STATE  (s_upper, s_lower)
     YIELDS EVENTS [:UpM(ev, hdr3); DnM(ev1, hdr2):]
        AND STATE  (s1_upper, s1_lower)

* THM ComposeUpSplit3
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower:State_lower.
  ∀s_upper,s1_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧ Upper = Fun(upper)
   ⇒ RECONFIGURING LAYER Lower
        FOR EVENT  UpM(ev, hdr)          AND STATE  s_lower
     YIELDS EVENTS [:DnM(ev1, hdr1); UpM(ev, hdr2):]
        AND STATE  s1_lower

      ∧ RECONFIGURING LAYER Upper
          FOR EVENT  UpM(ev, hdr2)       AND STATE  s_upper
       YIELDS EVENTS [:UpM(ev, hdr3):]   AND STATE  s1_upper

   ⇒ RECONFIGURING LAYER Upper::Lower
        FOR EVENT  UpM(ev, hdr)          AND STATE  (s_upper, s_lower)
     YIELDS EVENTS [:UpM(ev, hdr3); DnM(ev1, hdr1):]
        AND STATE  (s1_upper, s1_lower)

* THM ComposeDnLinear
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower:State_lower.
  ∀s_upper,s1_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev:Event.t.∀hdr,hdr1,hdr2:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧ Upper = Fun(upper)
   ⇒ RECONFIGURING LAYER Upper
        FOR EVENT  DnM(ev, hdr)          AND STATE  s_upper
     YIELDS EVENTS [:DnM(ev, hdr1):]     AND STATE  s1_upper

      ∧ RECONFIGURING LAYER Lower
          FOR EVENT  DnM(ev, hdr1)       AND STATE  s_lower
       YIELDS EVENTS [:DnM(ev, hdr2):]   AND STATE  s1_lower

   ⇒ RECONFIGURING LAYER Upper::Lower
        FOR EVENT  DnM(ev, hdr)          AND STATE  (s_upper, s_lower)
     YIELDS EVENTS [:DnM(ev, hdr2):]     AND STATE  (s1_upper, s1_lower)
```

```
* THM ComposeDnBounce
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower:State_lower.
  ∀s_upper,s1_upper,s2_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧  Upper = Fun(upper)
    ⇒ RECONFIGURING LAYER Upper
          FOR EVENT  DnM(ev, hdr)          AND STATE  s_upper
       YIELDS EVENTS [:DnM(ev, hdr1):]     AND STATE  s1_upper

      ∧ RECONFIGURING LAYER Lower
           FOR EVENT  DnM(ev, hdr1)        AND STATE  s_lower
         YIELDS EVENTS [:UpM(ev1, hdr2):] AND STATE  s1_lower

      ∧ RECONFIGURING LAYER Upper
           FOR EVENT  UpM(ev1, hdr2)       AND STATE  s1_upper
         YIELDS EVENTS [:UpM(ev1, hdr3):] AND STATE  s2_upper

    ⇒ RECONFIGURING LAYER Upper::Lower
          FOR EVENT  DnM(ev, hdr)          AND STATE  (s_upper, s_lower)
       YIELDS EVENTS [:UpM(ev1, hdr3):]    AND STATE  (s2_upper, s1_lower)

* THM ComposeDnSplit
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower:State_lower.
  ∀s_upper,s1_upper,s2_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3,hdr4:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧  Upper = Fun(upper)
    ⇒ RECONFIGURING LAYER Upper
          FOR EVENT  DnM(ev, hdr)          AND STATE  s_upper
       YIELDS EVENTS [:DnM(ev, hdr1):]     AND STATE  s1_upper

      ∧ RECONFIGURING LAYER Lower
           FOR EVENT  DnM(ev, hdr1)        AND STATE  s_lower
         YIELDS EVENTS [:UpM(ev1, hdr2); DnM(ev, hdr3):]
            AND STATE  s1_lower

      ∧ RECONFIGURING LAYER Upper
           FOR EVENT  UpM(ev1, hdr2)       AND STATE  s1_upper
         YIELDS EVENTS [:UpM(ev1, hdr4):] AND STATE  s2_upper

    ⇒ RECONFIGURING LAYER Upper::Lower
          FOR EVENT  DnM(ev, hdr)          AND STATE  (s_upper, s_lower)
       YIELDS EVENTS [:UpM(ev1, hdr4); DnM(ev, hdr3):]
          AND STATE  (s2_upper, s1_lower)

* THM ComposeDnSplit1
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower:State_lower.
  ∀s_upper,s1_upper,s2_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3,hdr4:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧  Upper = Fun(upper)
    ⇒ RECONFIGURING LAYER Upper
          FOR EVENT  DnM(ev, hdr)          AND STATE  s_upper
       YIELDS EVENTS [:DnM(ev, hdr1):]     AND STATE  s1_upper

      ∧ RECONFIGURING LAYER Lower
           FOR EVENT  DnM(ev, hdr1)        AND STATE  s_lower
         YIELDS EVENTS [:DnM(ev, hdr2); UpM(ev1, hdr3):]
            AND STATE  s1_lower

      ∧ RECONFIGURING LAYER Upper
           FOR EVENT  UpM(ev1, hdr3)       AND STATE  s1_upper
         YIELDS EVENTS [:UpM(ev1, hdr4):] AND STATE  s2_upper

    ⇒ RECONFIGURING LAYER Upper::Lower
          FOR EVENT  DnM(ev, hdr)          AND STATE  (s_upper, s_lower)
       YIELDS EVENTS [:UpM(ev1, hdr4); DnM(ev, hdr2):]
          AND STATE  (s2_upper, s1_lower)
```

```
* THM ComposeDnSplit2
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower:State_lower.
  ∀s_upper,s1_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧  Upper = Fun(upper)
    ⇒ RECONFIGURING LAYER Upper
         FOR EVENT  DnM(ev, hdr)          AND STATE  s_upper
      YIELDS EVENTS [:UpM(ev1, hdr1); DnM(ev, hdr2):]
         AND STATE  s1_upper

       ∧ RECONFIGURING LAYER Lower
           FOR EVENT  DnM(ev, hdr2)       AND STATE  s_lower
        YIELDS EVENTS [:DnM(ev, hdr3):]   AND STATE  s1_lower

    ⇒ RECONFIGURING LAYER Upper::Lower
         FOR EVENT  DnM(ev, hdr)          AND STATE  (s_upper, s_lower)
      YIELDS EVENTS [:DnM(ev, hdr3); UpM(ev1, hdr1):]
         AND STATE  (s1_upper, s1_lower)

* THM ComposeDnSplit3
  ∀State_lower,State_upper,Type_lower,Type_upper,Type_Lower,Type_Upper:TYPES.∀Lower:Type_Lower.
  ∀Upper:Type_Upper.∀lower:Type_lower.∀upper:Type_upper.∀s_lower,s1_lower:State_lower.
  ∀s_upper,s1_upper:State_upper.∀Headers,Arguments:TYPES.∀ls:View.local.∀vs:View.state.
  ∀ev,ev1:Event.t.∀hdr,hdr1,hdr2,hdr3:Headers.∀args:Arguments.
    Lower = Fun(lower)   ∧  Upper = Fun(upper)
    ⇒ RECONFIGURING LAYER Upper
         FOR EVENT  DnM(ev, hdr)          AND STATE  s_upper
      YIELDS EVENTS [:DnM(ev, hdr1); UpM(ev1, hdr2):]
         AND STATE  s1_upper

       ∧ RECONFIGURING LAYER Lower
           FOR EVENT  DnM(ev, hdr1)       AND STATE  s_lower
        YIELDS EVENTS [:DnM(ev, hdr3):]   AND STATE  s1_lower

    ⇒ RECONFIGURING LAYER Upper::Lower
         FOR EVENT  DnM(ev, hdr)          AND STATE  (s_upper, s_lower)
      YIELDS EVENTS [:DnM(ev, hdr3); UpM(ev1, hdr2):]
         AND STATE  (s1_upper, s1_lower)

*************************************************************************
***                  Compression Theorems                             ***
*************************************************************************

* THM Compress
∀State,TypeL,Type_l:TYPES.∀L:TypeL.∀l:Type_l.∀s,s1:State.∀Headers,Arguments:TYPES.
 ∀ls:View.local.∀vs:View.state.∀ev:Event.t.∀hdr,hdr1:Headers.
 ∀compress,expand:Headers -> Headers.∀args:Arguments.
    L = Fun(l)
    ⇒ RECONFIGURING LAYER L
         FOR EVENT  DnM(ev, hdr)          AND STATE  s
      YIELDS EVENTS [:DnM(ev, hdr1):]     AND STATE  s1

    ⇒ RECONFIGURING LAYER L   WRAPPED WITH COMPRESSION
         FOR EVENT  DnM(ev, hdr)          AND STATE  s
      YIELDS EVENTS [:DnM(ev, compress hdr1):]
         AND STATE  s1

* THM Expand
∀State,TypeL,Type_l:TYPES.∀L:TypeL.∀l:Type_l.∀s,s1:State.∀Headers,Arguments:TYPES.
 ∀ls:View.local.vs:View.state.∀ev:Event.t.∀hdr,hdr1:Headers.
 ∀compress,expand:Headers -> Headers.∀args:Arguments.
    L = Fun(l)
    ⇒ RECONFIGURING LAYER L
         FOR EVENT  UpM(ev, expand hdr)   AND STATE  s
      YIELDS EVENTS [:UpM(ev, hdr1):]     AND STATE  s1

    ⇒ RECONFIGURING LAYER L   WRAPPED WITH COMPRESSION
         FOR EVENT  UpM(ev, hdr)          AND STATE  s
      YIELDS EVENTS [:UpM(ev, hdr1):]     AND STATE  s1

*************************************************************************
***                END Reconfiguration Theory                         ***
*************************************************************************
```