Automated Fast-Track Reconfiguration of Group Communication Systems

Christoph Kreitz

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

Abstract. We present formal techniques for improving the performance of modular communication systems. For common sequences of operations we identify a fast-path through a stack of communication protocols and reconfigure the system's code accordingly. Our techniques are implemented by tactics and theorems of the NuPRL proof development system and have been used successfully for the reconfiguration of application systems built with the Ensemble group communication toolkit.

1 Introduction

Due to the wide range of safety-critical applications [1], the development of secure and reliable communication systems has become increasingly important. Many communication protocols have been developed to ensure a variety of properties in a broad number of environments. To maximize clarity and code re-use, systems are often divided into modules that correspond to individual protocols and can be combined into *stacks of protocol layers*, as illustrated in Figure 1. But in the restricted context of a particular application these modular systems turn out to be less efficient than monolithic implementations, as they contain redundant and unused code and require communication *between* the individual modules. Furthermore they have to add headers to a sender's message to indicate how the layers in the receiver's stack needs to be activated, although typical data messages activate only a few protocols at all. Thus the costs of modularity are unnecessarily large execution times and increased net loads.

By analyzing common sequences of operation, i.e. typical messages and the normal status of the communication system, one can identify a *fast-track* through a protocol stack and *reconfigure* the system's code accordingly. Experiments [8,7] have shown that dramatic efficiency improvements and speedups of factor 30–50 can be achieved by partial evaluation, elimination of dead code and communication between layers, compression of message headers, and delayed state updates. Fast-track optimizations, however, are not supported by current compilers. They cannot be done by hand either, as they require a deep understanding of the system's code and, due to the code size of typical applications, contain a high risk of error. Besides, they must be done after configuring the application system and cannot be provided a priori. Therefore it is necessary to develop *formal tools* for an automated reconfiguration of networked application systems.

W.R. Cleaveland (Ed.): TACAS/ETAPS'99, LNCS 1579, pp. 104–118, 1999.

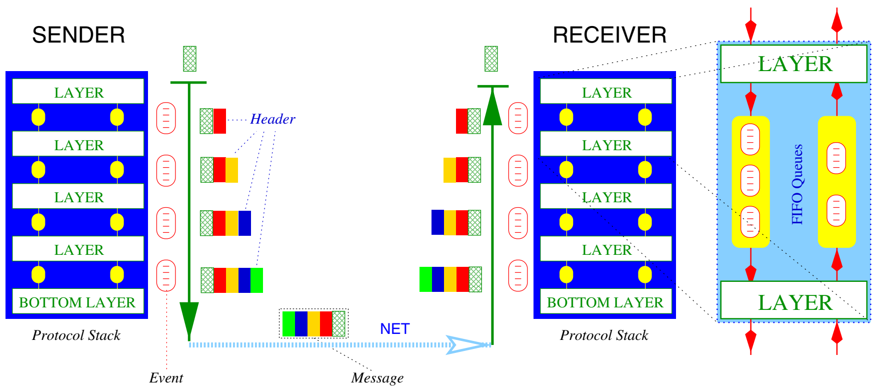

Fig. 1. Protocol stacking architecture: *Layers are linked by FIFO event queues*

To overcome the formalization barrier, which had prevented formal tools for analyzing software from being used to maximum benefit, we have linked Ensemble [7], a flexible group communication toolkit, to NuPRL [3], a proof system for mathematical reasoning about programs. The resulting *logical programming environment* [10] is capable of formal reasoning about the actual Ensemble code and provides the infrastructure for verifying important system properties [9] and for applying formal optimization techniques to the system's code.

In this paper we will describe the formal techniques for a reconfiguration of modular group communication systems that we have implemented within the logical programming environment. These techniques support both the developer and the user of a group communication toolkit. The former has the expertise to identify the path of a typical application message through the code of an individual protocol layer, i.e. when the system is in its regular state and does not have to deal with message partitioning, retransmission, buffering, synchronization, etc. We provide tactics for assumption-based symbolic evaluation of the code, which a system developer can use interactively to generate optimized pieces of code for each layer. Since these reconfigurations are independent from the application, they can be included in the distribution of the group communication toolkit. For the user of the toolkit we provide tools for reconfiguring the code of application protocol stacks and compressing the headers of common messages. Instead of using conventional rewrite techniques, which would not scale up very well, these tools are based on composing formal theorems about the results of reconfiguring individual protocol layers and are *fully automated*.

In section 2 we will briefly resume Ensemble, NuPRL, and the logical programming environment on which all our techniques are based. Section 3 discusses tactics for the reconfiguration of individual protocol layers. The theorem-based reconfiguration of protocol stacks is the topic of Section 4. Section 5 deals with header compression while Section 6 describes how to convert the logical reconfigurations into executable code. Finally we discuss the interface between the protocol stack and the application that uses it in section 7. For additional details not covered in this paper we refer to our technical report [12].

2 The Logical Programming Environment for Ensemble

Ensemble [7] is the third generation of group communication systems that aim at securing critical networked applications. The first system, Isis [2], became one of the first widely adopted technologies and found its way into many safety-critical applications. Horus [16], a modular redesign of Isis, introduced a protocol stacking architecture and techniques for reconfiguring a stack of protocol layers [8]. Reconfiguring Horus protocol stacks, however, is difficult and error prone, as the system is written in C and complex to reason about. Concerns about the reliability of such a technology base led to the implementation of Ensemble [6,7], which is based on Horus but coded almost entirely in Ocaml [14], a member of the ML language family. Due to the use of ML, Ensemble is one of the most scalable, portable, and also fastest existing reliable multicast systems. But the main reason for choosing Ocaml was to enable logical reasoning about Ensemble's code [5,7] and to apply formal methods for reconfiguring the system and verifying its properties.

The NuPRL *proof development system* [3] is a mathematical framework for reasoning about programs and program transformations. *Proof tactics* can be tailored to follow the particular style of reasoning in distributed systems. Program optimizations can be achieved by applying *rewrite tactics*, which reconfigure the code of a given application. NuPRL's formal calculus, *Type Theory*, includes formalizations of the fundamental concepts of mathematics, programming, and data types. It contains a functional programming language similar to the core of ML. The NuPRL system supports interactive and tactical formal reasoning, user-defined language extensions, program evaluation, and an extendable library of verified knowledge. These features make it possible to represent Ensemble's code and its specifications in NuPRL in order to use the system as a logical programming environment for the development of group communication systems.

The formal link between Ensemble *and* NuPRL has been established by converting the code of Ensemble into terms of NuPRL's logical language with the same meaning. In [11, Section 3] we have developed a *type-theoretical semantics* of Ocaml that is faithful with respect to the compiler and manual [14]. As Ocaml offers many features that are not used in Ensemble's code but might cause complications in a rigorous formalization, we have restricted the formalization to the subset of Ocaml required for implementing finite state-event systems, i.e. the functional subset together with simple imperative features. Our formalization was *implemented* by using NuPRL's definition mechanism: each Ocaml language construct is represented by a new NuPRL term that expresses its formal semantics. This *abstraction* is coupled with a *display form* that gives the formal representation the same outer appearance as the original code. Thus a NuPRL term represents both the text of an Ocaml program and its semantics.

We also have developed a formal *programming logic* for Ocaml by describing rules for reasoning about Ocaml expressions and for symbolically evaluating them [11, Section 4]. We have implemented these rules as NuPRL tactics, which proves them correct with respect to the type-theoretical semantics of Ocaml. As a result, formal reasoning about Ensemble within NuPRL can be performed entirely at the level of Ocaml programs, while program transformations always preserve the “Ocaml-ness” of terms. This enables system experts to reason formally about Ocaml-programs without having to understand type theory.

Finally, we have created tools that convert Ocaml programs into their formal NuPRL representations and store them as objects of NuPRL’s library [11, Section 5]. Using these tools we are able to translate the complete Ocaml-code of the Ensemble system into NuPRL (and vice versa) and to keep track of modifications and upgrades in Ensemble’s implementation.

3 Tactic-based Reconfiguration of Protocol Layers

A protocol layer is an independent implementation of a communication protocol that does not make any assumption about a sent or received message. It has to ensure properties such as FIFO transmission, total message ordering, virtual synchrony, group membership, etc. Although the essential algorithms can be expressed as simple state-event machines, most of the actual code has to deal with unusual situations that do not affect a typical data message. Consequently, a system developer who implements or maintains the communication toolkit can easily identify the assumptions that the code of each layer makes about common sequences of operation. This makes it possible to reconfigure the protocol layers independently. To support such a reconfiguration of protocol layers, we have developed three basic program transformation mechanisms that rewrite the code of a layer under a set of assumptions in a correctness preserving way.

Symbolic evaluation and function inlining simplifies the code of a protocol layer in the presence of constants or function calls. In a functional language such as Ocaml both techniques correspond to β-reduction. As our logical programming environment provides tactics that represent the reduction rules for each reducible Ocaml-expression, all formal rewrite steps operate on the level of the programming language and are guaranteed to be correct with respect to the type-theoretical semantics of Ocaml. To simplify an interactive application of these reductions we have summarized all reduction rules into a single evaluation tactic `Red`. It searches (top-down) for the first reducible subterm of an expression and reduces it. If given a subterm address as additional argument, it restricts the search accordingly. Thus it supports user-controlled simplifications of the code that allow a finer tuning than standard partial evaluation strategies, as the system developer can decide which reductions are actually meaningful.

Context-dependent simplifications help extracting the relevant code from a protocol layer. They have to trace the code-path of events that satisfy the assumptions about the common case and isolate the corresponding pieces of code. Since assumptions are usually formulated as equalities, this can be done by applying equality substitutions followed by reductions of the affected code pieces. We have developed a tactic `UseHyp`, which performs these steps automatically for a given assumption. Its implementation is straightforward, as the NuPRL system supports equality reasoning and the management of hypotheses.

```
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
```

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

Specialized transformation tactics take advantage of common structures in the code of protocol layers. For the sake of clarity, message events are often separated according to their direction (outgoing or incoming), type (sending, broadcasting, or others), and further criteria. Separate message handlers are applied to each case but the separation itself follows a common pattern. A detailed analysis of this common structure makes it possible to write a tactic that rewrites the code of a layer until the relevant message handler is isolated. In contrast to the previous techniques this tactic does not require search but applies a fixed series of rewrite steps. Furthermore, it may use transformations that cannot be represented by partial evaluation, as they are based on η-reductions, distributive laws, and other "undirected" equalities. Applying a specialized transformation tactic as a first step drastically simplifies the reconfiguration process for the system developer: it performs the tedious initial steps automatically and reduces the size of the code to be reconfigured interactively to a fraction of the original code.

We have developed a tactic `RedLayerStructure`, which is based on the common structure of Ensemble's protocol layers. Figure 2 shows the typical structure such a layer l. Essentially, it consists of two functions. The function `init` initializes the state of the layer according to a given view state `vs` and local state information `ls`, while `hdlrs` 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 l into the stack. Ensemble distinguishes five sub-handlers for up- and down-going regular events (`up,dn`), events with no headers (`upnm,dnnm`), and local events (`uplm` or `dnlm`). The layer l itself is created from `init` and `hdlrs` through the function `Layer.hdr`, which glues the five sub-handlers together. This makes it possible to convert the implementation of a layer into a functional, imperative, or threaded version while keeping a single reference implementation.

To reconfigure a the layer l we state the assumptions about common events and layer states and optimize the event handler of l, starting with the expression

let ($\mathtt{s}_{\mathtt{init}}$,hdlr) = convert Functional l args (ls, vs) in hdlr(s_l, *event*)

where `convert` generates the functional version of l, which consists of the initial state $\mathtt{s}_{\mathtt{init}}$ and the event handler `hdlr`. s_l is the current state of l and *event* an event of the form `UpM(ev,hdr)` or `DnM(ev,hdr)`. The assumptions about the common case are stated in the form of equations that characterize the type of the event `ev` (send or broadcast), the structure of the header `hdr` (full, no, or local header), and the state s_l. For the sake of clarity we suppress irrelevant details by using the following formal abbreviation for the above expression:

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

As first step in a reconfiguration we use the tactic `RedLayerStructure`. It unfolds the formal abbreviation and evaluates `convert Functional` l `args (ls,vs)`, which leads to rewriting `Layer.hdr init`$_l$ `hdlrs`$_l$ `args (ls,vs)`. After applications of reduction rules, laws about nested let-abstractions, and η-reductions the tactic isolates the relevant handler from `up_hdlr` ... `dnnm_hdlr` by matching *event*, which has the form `UpM(ev,hdr)` or `DnM(ev,hdr)` where `hdr` is `Full(hdr,abv)`, `NoMsg`, or `Local hdr`, against a case expression in the code of `Layer.hdr`. As result we get the code of the relevant message handler.

Afterwards we apply the tactic `UseHyp` to use assumptions whenever they help to eliminate branches of a conditional or a case expression and apply top-level reductions (i.e. the tactic `Red`) to make other assumptions applicable. We continue with controlled reductions until no more optimization is meaningful.

These three basic tactics thus lead to a simple methodology for reconfiguring protocol layers. Except for the decision when to finish the transformation, which requires some insight into the implementation and can only be made by a system developer, it can be almost completely automated. We have used it successfully to reconfigure 25 of Ensemble's 40 protocol layers for the four most common kinds of events, i.e. incoming and outgoing send- or broadcast messages. 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.

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 = NoHdr`), and (3) in the state `s_bottom` there is no record of 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` makes the assumptions explicit and then evaluates the remaining program expression as described above.

```
ASSUME 1. getType ev = ECast
       2. s_bottom.failed.(getPeer ev) = false
⊢ (match (getType ev, NoHdr) with
      ((ECast | ESend), NoHdr)  -> ......
     | (ECast, Unrel)           -> ......
                          :
     |                    :
  )  (s_bottom, Fqueue.empty)
```

where '......' indicates that details of the code are temporarily hidden from the display. We now call `UseHyp 1`, which leads to an evaluation of the first case of the case expression and eliminates all the other cases from the code.

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

Next, we use the assumption `s_bottom.failed.(getPeer ev) = false` and evaluate of the first case of the conditional by calling `UseHyps 2`. This results in

```
⊢ (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)`, are explicitly stated. Under the the given assumptions we know now

```
hdlr_b(s_bottom,UpM(ev,Full(NoHdr,hdr)))  =  (s_bottom,[:UpM(ev,hdr):])
```

where $\mathtt{hdlr}_b$ denotes the event handler of the bottom layer and `[:UpM(ev,hdr):]` abbreviates `Fqueue.add UpM(ev,hdr) Fqueue.empty`. This means that the state of the layer remains unchanged while the original message is passed to the next layer after the header `NoHdr` has been stripped off.

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, evaluation, and verified laws, 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. Formally, we have to prove

```
let (s_init,hdlr) = convert Functional l args (ls,vs) in  hdlr(s_l, event)
=  (s'_l, [:out-events:])
```

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 our reconfiguration mechanisms and the logical inference rules of the NuPRL proof development system. It is easy to write *proof tactics* that perform exactly the same steps on the left hand side of an equation as our *reconfiguration* tactics `Red`, `UseHyps`, and `RedLayerStructure` 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 considerable 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. Since the tactic is guaranteed to succeed, it runs as a background process after a reconfiguration has been finished.

4 Theorem-based Protocol Stack Reconfiguration

In contrast to individual layers, 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 designer of an application system who uses a group communication toolkit must also be given a tool that creates a fast-track reconfiguration of the application system *automatically*.

It is easy to see that tactic-based rewrite techniques are not appropriate for this purpose, as they require interaction and expertise about the code of the layers and of the mechanism for composing layers. Furthermore they do not scale

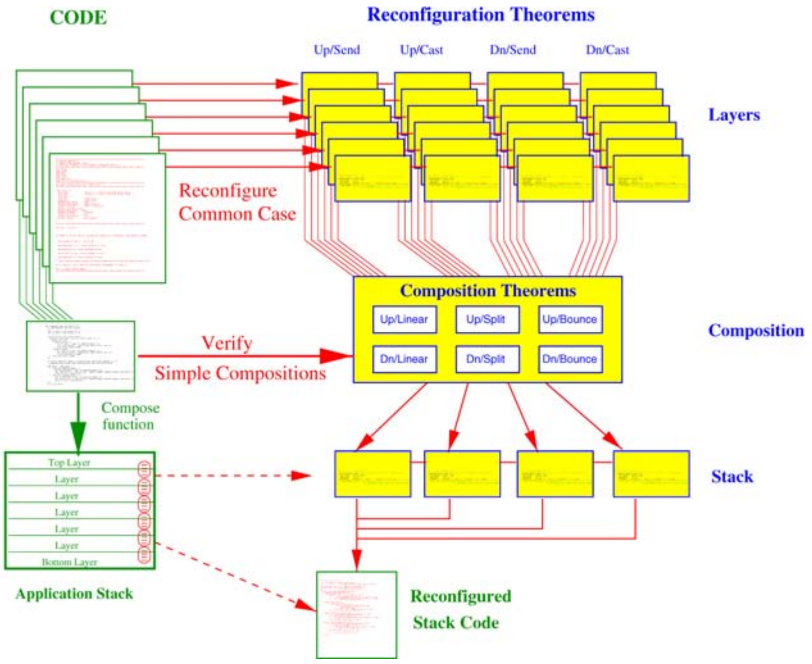

Fig. 3. Reconfiguration methodology: composing reconfiguration theorems

up very well: since messages may create additional events on their path through a protocol stack the reconfiguration tactics would have to deal with the entire code of the stack at once, which means that each rewrite step must operate on extremely large terms (representing more that 10000 lines of code).

On the other hand, a fast-path through a protocol stack is characterized by the fact that events pass through the stack without generating more than one or two additional events. Thus it is possible to *derive* the result of passing a common event through a protocol stack from already known reconfiguration results for the individual protocol layers: instead of having to symbolically evaluate the complete code from scratch we compose the individual reconfiguration results according to our knowledge about the code for layer composition.

Technically, we do this by *composing formal theorems*, as illustrated in Figure 3.

– For each protocol layer we prove *reconfiguration theorems* about the result of reconfiguring its code for the most common types of events, i.e. up- and down-going send- and broadcast messages. Since these theorems only depend on the implementation of the protocol layers but not on the particular application system, they can be proven once and for all and be included in the distribution of the communication toolkit. For Ensemble we use the equivalence theorems that are generated automatically after finishing the reconfiguration of a layer, as discussed in section 3
– For composing fast-paths through individual layers into a fast-path through a protocol stack we prove *composition theorems* about common combinations of fast-paths, such as *linear traces* (where an event passes through a layer), bouncing events, and messages that cause several events to be emitted from a

```
 THM ComposeDnLinear
   RECONFIGURING LAYER Upper FOR EVENT    DnM(ev, hdr)    AND STATE s_up
                             YIELDS EVENTS [:DnM(ev, hdr1):] AND STATE s1_up
 ∧ RECONFIGURING LAYER Lower FOR EVENT    DnM(ev, hdr1)   AND STATE s_low
                             YIELDS EVENTS [:DnM(ev, hdr2):] AND STATE s1_low
⇒  RECONFIGURING LAYER Upper|||Lower FOR EVENT DnM(ev, hdr)   AND STATE (s_up, s_low)
                                 YIELDS EVENTS [:DnM(ev, hdr2):] AND STATE (s1_up, s1_low)
```

Fig. 4. Reconfiguration theorem for linear down traces

layer (splitting) – both for up- and down-going events. While the statements of these theorems often appear trivial, their proofs are rather complex as we have to reason about the actual code of layer composition and to perform all steps that would usually occur *during* a reconfiguration. By proving the composition theorems we express the logical laws of layer composition as *derived inference rules* and remove a significant deductive burden from the reconfiguration process: reconfiguring composed protocol layers can now be done by theorem application in a *single* inference step where a tactic-based reconfiguration would have to execute hundreds of elementary steps.
Figure 4 presents a reconfiguration theorem for composing down-going linear traces in ENSEMBLE. Assuming that a down-going event through the layers `Upper` and `Lower` yields a queue consisting of a single down-event and possibly modifies the state of these layers, we prove that sending the event through the composed stack `Upper|||Lower` (where ||| is ENSEMBLE's composition function) does the obvious: states will be updated independently while the event is first modified by the upper layer and then by the lower layer.

– Using the above theorems we can generate and prove *reconfiguration theorems for a given protocol stack*. To create the statement of such a theorem we consult the theorems about layer reconfigurations for the corresponding events and compose them as described by the composition theorems. Starting with the top of the stack we match incoming and outgoing events of the theorems for adjacent layers to determine the structure of the event that must enter the stack and the result of passing it through the stack. The states of the layers will be composed into tuples of states and the assumptions will be accumulated by conjunctions.
To prove the stack reconfiguration theorem we use the information that we had gained while stating it. We instantiate the reconfiguration theorems of the layers in the stack with the actual event that will enter them. We then apply step by step the appropriate composition theorems to compose the fast-paths through the stack until the result is identical to the original statement of the theorem. Both proof steps are very easy to implement as they only require us to apply instantiated versions of already proven theorems.
For ENSEMBLE we have developed a tactic `CreateReconfiguredStack` that, given a list of layer names, generates the reconfiguration theorem, proves it correct, and stores it under a unique name. Since all of these steps are completely automated the tactic does not require any user interaction but can instead integrated into ENSEMBLE's configurator.

– From the logical reconfiguration theorems we finally generate OCAML code for a modified protocol stack that can be used to replace the original stack.

We will discuss *code generation* in Section 6 after describing how to optimize a reconfigured stack by *header compression* (see Section 5).

Theorem-based layer composition leads not only to fully automated reconfiguration techniques but also to a much clearer style of reasoning as we raise the abstraction level of program transformations from programming language expressions to reasoning about modules. It also improves the performance of the reconfiguration process, which requires only a few steps for each protocol layer passed by an event and thus scales up very well. Finally system updates can be handled much easier: the modification of a layer's code only requires reproving the reconfiguration theorems for this particular layer while the reconfiguration of the protocol stack will remain unaffected or is re-executed automatically.

Example 2. To reconfigure the stack Pt2pt ||| Mnak ||| Bottom for outgoing send-messages, CreateReconfiguredStack consults the following reconfiguration theorems.

```
THM Pt2ptReconfDnMESend_verif
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
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 BottomReconfDnMESend_verif
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, they have to be composed 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 is matched against the incoming event of Mnak and the variable hdr in theorem MnakReconfDnMESend_verif is 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 the assumptions are composed by conjunction. As a result CreateReconfiguredStack creates and proves the following reconfiguration theorem.

```
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
               )
```

5 Header Compression

After reconfiguring a protocol stack we know exactly which headers are added to a typical data message by the sender's stack and how the receiver's stack processes these headers in the respective layers. A message that goes through the

```
 THM Compress
    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
    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. 5. Compression and expansion theorems for down/up-traces

fast-path obviously does not activate many protocol layers. Consequently, most of the added headers indicate that the layer has not been active. Such information does not have to be transmitted over the net if we encode the fact that the message has gone through the fast-path. Transmitting only the relevant headers will reduce the net load and improve the overall efficiency of communication.

A straightforward method for eliminating irrelevant headers from a transmitted message is to generate code for *compressing* and *expanding* headers and to insert it between the protocol stack and the net. Compression removes all the constants from a header and leaves only the information that may vary. In the stack Pt2pt ||| Mnak ||| Bottom from example 2, for instance, an outgoing send-message receives the header

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

This header contains keyword constants like Full, NoHdr, and Data that do not carry essential information. Without loss of information it can be compressed to

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

To create the code for compression and expansion, we consult the reconfiguration theorems for received common messages and look at the structure of the headers of incoming events. Compression matches a header against this pattern and generates a new header only from the free variables in the pattern while removing all the constants. Headers that do not match such a pattern will not be changed. Header expansion simply inverts compression. Both programs can be generated automatically after a reconfiguration. For the stack Pt2pt ||| Mnak ||| Bottom, for instance, we get the following two programs.

```
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
```

Header compression can easily be integrated into the reconfiguration process. For this purpose we reconfigure the code of a protocol stack *after wrapping it with compression*. By doing so we generate an optimized stack that directly operates on compressed messages. Again we propose a theorem-based approach: since we already know how to reconfigure a regular protocol stack we prove generic *compression* and *expansion theorems* that describe the outcome of reconfiguring a wrapped stack in terms of the results of reconfiguring the regular stack.

Figure 5 presents the compression and expansion theorems for Ensemble. They describe the obvious effect of applying Ensemble's function wrap_hdr to a stack L and functions compress and expand, which we formally abbreviate by L WRAPPED WITH COMPRESSION. Proving the theorems removes another burden from the reconfiguration process: we can now make the transition from a reconfigured ordinary stack to its wrapped version in a single inference step.

Based on compression and expansion theorems a reconfiguration of wrapped protocol stacks follows the same methodology as before. To generate the statement of the reconfiguration theorem, we compose the reconfiguration theorems for its layers and then compose the result with the compression and expansion theorems. To prove it, we first insert the result of a regular fast-track reconfiguration. For outgoing messages we then transform emitted headers into the form '`compress` *hdr*' and apply the theorem Compress. For received messages we transform incoming headers into '`expand` *hdr*' and apply the theorem Expand.

For Ensemble we have developed a tactic CreateCompressedStack that performs all these steps automatically. For outgoing send-messages through the stack Pt2pt ||| Mnak ||| Bottom (c.f. example 2) wrapped with compression, for instance, it creates and proves the following reconfiguration theorem.

```
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
              )
```

6 Code Generation

The reconfiguration theorems for protocol stacks describe how to handle common events in a much more efficient way. In order to use these results in a running application system we have to convert the theorems into Ocaml-code that deals with *all* possible cases. For this purpose we introduce a "switch" that identifies the common case and sends fast-path messages to the reconfigured code while passing all other messages to the code of the original protocol stack.

To convert a reconfiguration theorem into pieces of code we transform the reconfiguration results described by them, i.e. the modified states of the protocol stack and the queue of events to be emitted, into the code that creates these results. Modified states of the form s[.f←e] are converted into an assignment s.f<-e. No assignments are generated for unmodified states. A queue [:$\mathtt{ev}_1$; ..; $\mathtt{ev}_n$:] of emitted events is converted into a sequence of calls of event handlers for up- and down-going events. 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.

To generate the switch we consider all the reconfiguration theorems of the protocol stack at once and create a case expression that separates the corresponding pieces of code. Usually we distinguish four fundamental cases: incoming and outgoing send- and broadcast messages. Assumptions of the reconfiguration theorems that do not deal with the direction or the type of an event are converted into a conditional or into another case expression if free variables occur. As an optimization we evaluate assumptions about generated events and eliminate those which are simple variations of other assumptions or evaluate to `true`. Events that fit one of four patterns and satisfy the conditions of that case will be directed to the code of the corresponding generated event handler. All other events will be passed to the event handler of the original stack.

Using these insights we have developed a tactic `ReconfiguredBody`, which automatically generates the complete OCAML-code for the optimized protocol stack after from the reconfiguration theorems for the stack. For the three-layer stack `Pt2pt`|||`Mnak`|||`Bottom` (c.f. example 2) wrapped with compression, for instance, it creates the following optimized stack.

```
let opt_stack state (ls,vs) =
 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) in
 let orig_stack = wrap_hdr compress expand (Pt2pt.l ||| Mnak.l ||| Bottom.l)  in
  let s,hdlr           = orig_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   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)                   -> ...
    | (UpM(ev, OptSend(seqno, hdr)), ESend)  -> ...
    | (UpM(ev, OptCast(seqno, hdr)), ECast)  -> ...
    | _                                        ->  hdlr ((s_pt2pt,s_mnak,s_bottom),event)
   in
       (s,opt_hdlr)
```

After the code for the optimized stack has been generated as a NuPRL object we prove it to be equivalent to the original stack. We then export the code into the OCAML environment source file and compile it into executable code.

7 The Application Interface

The protocol stacking architecture depicted in figure 1, which places the application on top of the stack, is a simplified model of the real architecture of efficient communication systems. As reliable group communication has to deal with many aspects that are not related to the application, application messages should not have to pass through the complete protocol stack but only through the protocols that are necessary for handling data. Therefore ENSEMBLE connects the application to a designated layer `partial_appl` *within* the stack (see left hand side of figure 6). Protocols that deal with the management of the group, e.g. with stability, merging, leaving, changing groups, virtual synchrony, etc. reside on top of this layer and are not used by application messages.

While this refined architecture improves the efficiency of ENSEMBLE it complicates a fully formal reconfiguration of its protocol stacks, because the interaction between the `partial_appl` layer and the application does not rely on events anymore. Instead, application messages are processed by two functions `recv_send` and `recv_cast`, which in turn provide a list of actions that are converted into events. From the viewpoint of communication, the application is just a part of `partial_appl` and we have reason about the effects of `recv_send` and `recv_cast` to create a fast-path through `partial_appl`. These two functions link up- and down-going events and may also initiate many new events at once.

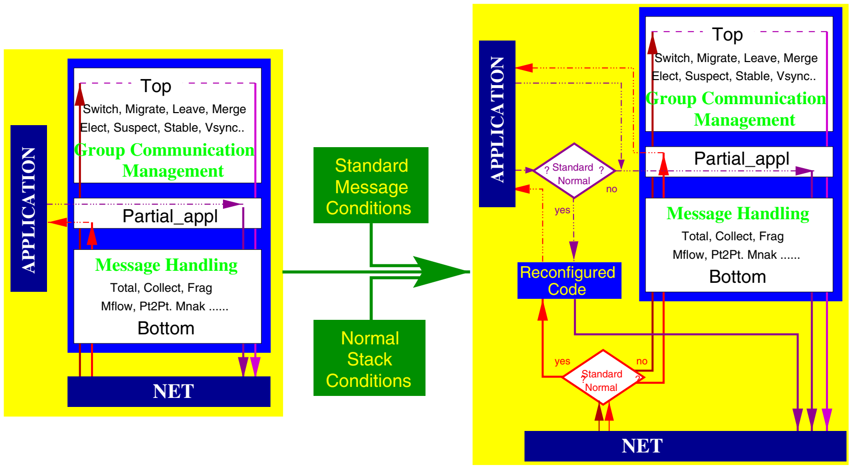
APPLICATION
Top
Switch, Migrate, Leave, Merge
Elect, Suspect, Stable, Vsync..
Group Communication Management
Partial_appl
Message Handling
Total, Collect, Frag
Mflow, Pt2Pt. Mnak ......
Bottom
NET
Standard Message Conditions
Normal Stack Conditions
APPLICATION
? Standard ? Normal
yes
no
Reconfigured Code
Top
Switch, Migrate, Leave, Merge
Elect, Suspect, Stable, Vsync..
Group Communication Management
Partial_appl
Message Handling
Total, Collect, Frag
Mflow, Pt2Pt. Mnak ......
Bottom
yes
no
? Standard ? Normal
NET

Fig. 6. Stack reconfiguration including the `partial_appl` layer

Since the number of emitted events is not fixed, we cannot create theorems about the *results* of reconfiguring `partial_appl`. Instead, we must directly create a reconfigured version of the *code* of `partial_appl`, and apply specialized tactics to compose the resulting code with the already optimized code for the remaining stack. This leads to a transformed application stack as illustrated in figure 6: the reconfigured protocol interacts directly with the application and is very efficient.

While it is comparably easy to *generate* the complete code of the reconfigured stack, proving it to be equivalent to the original one is more difficult. We cannot use the composition theorems from section 4 for describing the effects of composing `partial_appl` with the rest of the stack but have to use tactics that deal specifically with this situation. This makes a verification of the reconfigured stack more time consuming than a purely theorem-based approach but does not affect the reconfiguration process itself.

8 Conclusion

We have presented a variety of formal techniques for improving the performance of modular communication systems and applied them to networked systems built with the Ensemble group communication toolkit. They provide both interactive tools for a system developer, who uses expertise about the code to improve individual protocol layers, and fully automated reconfiguration mechanisms for a user of the communication toolkit, who designs application systems.

We have implemented our techniques as tactics of the NuPRL proof development system, which are based on an embedding of Ensemble's code into NuPRL's logical language. This guarantees the correctness of all optimizations with respect to the formal semantics of the code and enables us to use *theorem-based* rewriting, which raises the abstraction level of program transformations from expressions of the programming language to system modules. This leads to a much clearer style of reasoning and makes our reconfiguration techniques scale up very well. To our knowledge there is no other rigorously formal system that can reason about the complete code of realistic applications.

We have used our techniques to reconfigure the 22-layer protocol stack of a running application system, which resulted in significant improvements of the stack's performance. In the future we intend to refine our techniques and to integrate them into the distribution of the ENSEMBLE group communication toolkit. We also plan to add further reasoning capabilities to the logical programming environment in order to verify the code of ENSEMBLE's protocol layers [9] and protocol stacks. For this purpose we will integrate general deductive tools – such as extended type-analysis [4], first-order theorem proving [13], and inductive proof methods [15] – and develop proof tactics that are specifically tailored to ENSEMBLE's code. By combining all these techniques into a single environment we expect to create a software development infrastructure for the construction of efficient and reliable group communication systems.

References

1. K. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, 1997.
2. K. Birman & R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.
3. R. Constable, et. al., *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, 1986.
4. O. Hafızoğulları & C. Kreitz. A Type-based Framework for Automatic Debugging. Technical Report, Cornell University, 1998.
5. M. Hayden. Distributed communication in ML. Technical Report TR97-1652, Cornell University, 1997.
6. The ENSEMBLE distributed communication system. System distribution and documentation. `http://www.cs.cornell.edu/Info/Projects/Ensemble`
7. M. Hayden. *The Ensemble System.* PhD thesis, Cornell University, 1998.
8. M. Hayden & R. van Renesse. Optimizing layered communication protocols. Technical Report TR 96-1613, Cornell University, 1996.
9. J. Hickey, N. Lynch, R. van Renesse. Specifications and Proofs for Ensemble Layers. *TACAS'99*. This volume
10. C. Kreitz, M. Hayden, J. Hickey. A proof environment for the development of group communication systems. *CADE-15*, LNAI 1421, pp. 317–332, Springer, 1998.
11. C. Kreitz. Formal reasoning about communication systems I: Embedding ML into type theory. Technical Report TR97-1637, Cornell University, 1997.
12. C. Kreitz. Formal reasoning about communication systems II: Automated Fast-Track Reconfiguration. Technical Report TR98-1707, Cornell University, 1998.
13. C. Kreitz, J. Otten, S. Schmitt. Guiding Program Development Systems by a Connection Based Proof Strategy. *LoPSTR-5*, LNCS 1048, pp. 137–151. Springer, 1996.
14. X. Leroy. *The Objective Caml system release 1.07.* Institut National de Recherche en Informatique et en Automatique, 1998.
15. B. Pientka & C. Kreitz. Instantiation of existentially quantified variables in inductive specification proofs. *AISC'98*, LNAI 1476, pp. 247–258, Springer, 1998.
16. R. van Renesse, K. Birman, & S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

10.1007/b107031130008