A Proof Environment for the Development of Group Communication Systems

Christoph Kreitz Mark Hayden Jason Hickey

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

Abstract. We present a theorem proving environment for the development of reliable and efficient group communication systems. Our approach makes methods of automated deduction applicable to the implementation of real-world systems by linking the Ensemble group communication toolkit to the NuPRL proof development system.
We present tools for importing Ensemble's code into NuPRL and exporting it back into the programming environment. We discuss techniques for reasoning about critical properties of Ensemble as well as verified strategies for reconfiguring the Ensemble system in order to improve its performance in concrete applications.

1 Introduction

Group communication via computer networks is used in a wide range of applications [3]. Over the past years the development of a secure and reliable communications infrastructure has become increasingly important. But the current networks are inadequate to support safety-critical applications because considerable technical challenges have not been overcome yet.

First, there is the *performance cost* of modularity. To maximize clarity and code re-use, systems are divided into clean modules, which are designed to operate in a broad number of environments. But when modules are combined in a restricted context, much of the code becomes useless or redundant, leading to unnecessary large execution times. Secondly, there is the *secure implementation problem*: designing and correctly implementing distributed systems is notoriously difficult [3, 5]. While in principle it is possible to prove the correctness of theoretical algorithms [19, 18, 1, 20] it is very difficult to transform these idealizations into implementations that can actually be used in real systems. Finally, the *formalization barrier* prevents formal tools for checking software correctness from being used to maximum benefit. These tools are computationally costly and difficult to understand; even well understood type checking algorithms are often viewed as expensive. Few of these tools are integrated into software development environments, nor can they be flexibly and interactively invoked.

In this paper we address these problems by showing how to make methods of automated deduction applicable to the implementation of a real-world system. Our approach links Ensemble [11], a flexible group communication toolkit, to NuPRL [6], a proof system for mathematical reasoning about programs and for rewriting them into equivalent, but more efficient ones.

C. & H. Kirchner, ed. 15^{th} *International Conference on Automated Deduction*,
LNAI xxx, pp. **??**–**??**, Springer Verlag, 1998.

Because of the similarity between the core of OCAML [16], the implementation language of ENSEMBLE, and *Type Theory*, the logical language of NUPRL, we were able to translate the complete implementation of ENSEMBLE into NUPRL-terms and to apply proof tactics and verified program transformations to the *actual* ENSEMBLE *code*. This makes it possible to verify critical system properties and to improve its performance in particular applications. NUPRL thus becomes a *logical* programming environment for ENSEMBLE whose capabilities go beyond the usual type-checking and syntactical debugging capabilities of OCAML. It will eventually provide the software development infrastructure and a design methodology for constructing reliable and efficient group communication systems.

In Section 2 we will present the architecture of the logical programming environment, including a brief overview of ENSEMBLE and NUPRL. In Section 3 we describe the representation of the relevant subset of OCAML in Type Theory as well as the tools for importing ENSEMBLE's code into NUPRL and exporting it back into the OCAML environment. In Section 4 we discuss techniques for verifying system properties and in Section 5 we describe proof and rewrite tactics for a verified reconfiguration of ENSEMBLE in a given application-specific context.

2 Architecture of the Logical Programming Environment

The ENSEMBLE toolkit is the third generation of a series of group communication systems that aim at securing critical networked applications. The first system, ISIS [4], became one of the first widely adopted technologies in this area and found its way into Stock Exchanges, Air Traffic Control Systems, and other safety-critical applications. The architecture of HORUS [21], a modular redesign of ISIS, is based on stacking *protocol layers*, which can be combined almost arbitrarily to match the needs of a particular application. Despite its flexibility, HORUS is even 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.

However, reconfiguring HORUS protocol stacks is difficult and error prone because its layers are written in C and they are too complex to reason about. Concerns about the reliability of such a technology base for truly secure networked applications led to the implementation of ENSEMBLE [11, 12], which is based on HORUS but coded almost entirely in the high-level programming language OCAML [16], a member of the ML [9] language family with a clean semantics. 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. One of the main reasons for choosing OCAML, however, was to enable formal reasoning about ENSEMBLE's code within a theorem proving environment and to use deduction techniques for reconfiguring the system and verifying its properties.

Conceptually, each protocol stack is a finite IO-automaton, which could be handled by propositional methods. But reasoning about the *actual* ENSEMBLE *code* requires a theorem proving environment that is 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.

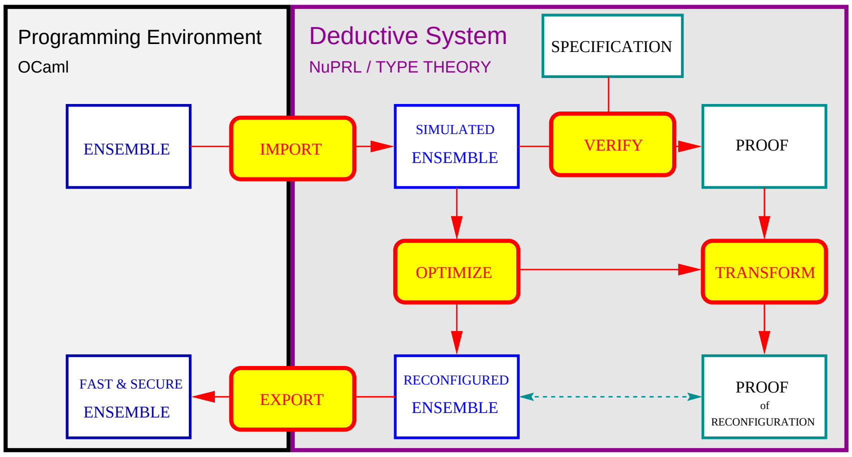

Fig. 1. Verifying and reconfiguring communication systems in NuPRL

The NuPRL proof development system [6] 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. Performance improvements can be achieved by applying *rewrite tactics*, which reconfigure the protocol stack for a particular application.

Because of its expressive formal calculus, NuPRL is well suited for building a reasoning environment for Ensemble. NuPRL's *Type Theory* 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* (LPE) for Ensemble.

Figure 1 illustrates our methodology for developing efficient and secure communication systems with the logical programming environment. In the first step a well-structured Ensemble protocol stack is be imported into the system, i.e. converted into NuPRL-terms. We can then apply verification tactics to prove critical protocol properties and system invariants. We can also apply optimization tactics to create a fast-track reconfiguration of the protocol stack that is guaranteed to have the same behavior as the original one. We can also prove this fact formally and transform a verification of the original system into one of the reconfigured stack. The latter is then exported back into the programming environment and used as improved and secured part of the application system.

Using this methodology we are able to address the three above-mentioned challenges. By building an environment that treats the actual Ensemble code, we substantially lower the formalization barrier. By providing tactics for reasoning about system properties we address the secure implementation problem. The performance costs of a protocol stack is drastically reduced by applying reconfiguration strategies and exporting their results into the OCAML environment. In the rest of this paper we shall discuss each of these aspects separately.

3 Embedding System Code into NUPRL

In order to enable formal reasoning about the code of an ENSEMBLE protocol stack, we have to convert OCAML programs into terms of the logical language of NUPRL that capture the semantics of these programs and vice versa. For this purpose we have provided a *type-theoretical semantics* for the programming language OCAML that is faithful with respect to the OCAML compiler and manual [16]. We have limited our formalization to the subset of OCAML that is used in the implementation of finite state-event systems like ENSEMBLE, i.e. the functional subset with simple imperative features. By doing this we avoided having to deal with aspects of the language that do not occur when reasoning about protocol stacks but cause unnecessary complications in a rigorous formalization.

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 constructs and rules for symbolically evaluating them. The rules were implemented as NUPRL tactics and are therefore correct with respect to the type-theoretical semantics of OCAML.

Finally, we have created tools that convert OCAML programs into their formal NUPRL representations and store them as objects of NUPRL's library. These tools are necessary to make the actual OCAML-code of an ENSEMBLE protocol stack available for formal reasoning within NUPRL and to keep track of modifications in ENSEMBLE's implementation.

As a result, formal reasoning within NUPRL can now 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 will always preserve the "OCAML-ness" of the terms they are dealing with. This enables system experts who are not necessarily logicians to formally reason about OCAML-programs without having to understand the peculiarities of the underlying theorem proving environment.

In the rest of this section we briefly describe the formalization of OCAML and the tools that translate between OCAML programs and their formal representations. For a full account we refer the reader to our technical report [14].

3.1 Formalizing OBJECTIVE CAML in Type Theory

OCAML [16] is a strongly typed, (almost) functional language, which has been extended with a module system and an object calculus. Its functional core is similar to the language of type theory, but it has a different syntax and contains many additional features.

Standard data types such as arrays, records, queues, etc. and their operations are predefined in OCAML but have to be represented by more fundamental constructs in type theory. In most cases, the formalization is straightforward. Arrays over some type T, for instance, are represented by pairs (lg,a) where $lg \in \mathbb{N}$ is the array's length and $a{:}\mathbb{N}\to T$ the component selector function. Records are represented by dependent functions and variant records by dependent products. But there are also language constructs whose representations are more complex.

Variable-sized Expressions. In contrast to type-theory, which requires terms to have a fixed number of subterms, OCAML allows expressions with arbitrarily many components. $\{l_1{=}v_1;..;l_n{=}v_n\}$, for instance, denotes a record that has the value v_i at field l_i. This expression can be represented by a function r that on input l_i yields the value v_i, but r can be *described* only through an iterated application of several NUPRL-abstractions: a representation of the empty record $\{\}$ by the constant function $\lambda l.()$ and a representation for record extension by a new field $l_{n+1}{=}v_{n+1}$. Appropriate display forms make sure that a term built from these abstractions is always displayed like the corresponding OCAML-record.

Thus formally OCAML language constructs are not associated with individual NUPRL-abstractions but with *term-generators*, i.e. meta-level programs that construct a NUPRL-term out of one or more abstractions.

Pattern Matching. A convenient feature of OCAML is the support of pattern matching in local abstractions, function definitions, and case-expressions. `match` *expr* `with` p_1`->`t_1`...`p_n`->`t_n, for instance, subsequently matches the expression *expr* against the patterns $p_1 \ldots p_n$. If matching succeeds with pattern p_i then the free variables of p_i will be instantiated in t_i and t_i will be evaluated.

Although a pattern looks like a conventional expression, it has an entirely different semantics. Computationally, a pattern p can be viewed as a matching function (or *matcher*) that takes an expression *expr* and a target t, analyzes the structure of *expr*, and returns an instance of t with the free variables of p instantiated. A pattern x, for instance, contains the free variable x and matches against any expression. Applying the matcher to *expr* and t results in $t[expr/x]$.

Since most patterns are constructed from other patterns we also need mechanisms for composing matchers. The paired pattern (p_1,p_2), for instance, expects to be matched against a pair (e_1,e_2). The term e_1 is handed together with t to the matcher p_1 while e_2 and the result of this matching is given to p_2.

Finally, we have to deal with the fact that matching may fail. Therefore, a matcher returns both a modified target expression and a boolean value.

Taking these aspects into account we have implemented a collection of abstractions and display forms for each pattern construct. The abstractions define the term structure and the semantics of a matcher. The display form describes its outer appearance and makes matcher terms look like the original pattern.

Example 1. The abstraction for paired patterns introduces a (higher-order) term with the name `Product__Match_Pair` and two formal subterms, the submatchers p_1 and p_2, which are separated by a semicolon. The right hand side of the abstraction formalizes the intuition of paired matchers. A display form causes the matcher to appear as simple pair.

$$\begin{aligned}
&\underline{\texttt{Product__Match_Pair}\{\}\texttt{(.}p_1\texttt{;.}p_2\texttt{)}}\\
&\qquad\equiv\ \lambda\texttt{expr,t. let e}_1\texttt{,e}_2\texttt{ = expr in}\\
&\qquad\qquad\qquad\texttt{let b,t' = (}p_1\texttt{ e}_1\texttt{ targ) in}\\
&\qquad\qquad\qquad\quad\texttt{if b then (}p_2\texttt{ e}_2\texttt{ t') else (false, t')}\\
&\underline{p_1\texttt{,}\,p_2}\ \equiv\ \texttt{Product__Match_Pair}\{\}\texttt{(.}p_1\texttt{;.}p_2\texttt{)}
\end{aligned}$$

Imperative Features. For the sake of efficiency OCAML supports imperative features such as assignments, compound statements, and loops, while the primitives of type theory are completely functional. But for formal *reasoning* about OCAML-programs, a representation of imperative features based on a copy semantics is sufficient. Nevertheless, a complete formalization of imperative behavior would require a general model for managing reference variables.

Fortunately, ENSEMBLE's architecture is essentially a finite state-event system. Imperative features only affect the state of a protocol layer and the queue of events that links two layers. Thus we can use a simpler model and represent imperative assignments by functions that modify these two variables. Compound statements and loops are represented by function composition and recursion.

Modules and Object Declarations. OCAML-declarations of user-defined types and functions introduce a name for a new object and bind it to a given expression. Technically, declarations are instructions for the programming environment. In NUPRL they correspond to meta-level programs that add new abstractions and display forms to the library. These *object-generators* also deal with name space management, references to other user-defined objectd, overloading, and similar issues related to the environment of a program.

Similarly, OCAML modules can be considered to be a means for structuring the code. This allows, for instance, using the same name for different functions in different modules and supports a clear and uniform presentation of the protocol layers of ENSEMBLE. In NUPRL modules have to be mapped onto the flat name space of the library. Module declarations thus affect how object-generators determine the names of generated objects or of objects referred to by an identifier.

3.2 Conversion Algorithms

Given the formal embedding of OCAML the methodology for importing and exporting system code into NUPRL is straightforward. We have to analyze the syntax of OCAML-programs and create the corresponding term- and object generators. These generators will create appropriate terms and store all declared functions and types as new abstractions in NUPRL's library. In order to ensure faithfulness wrt. the OCAML programming environment we chose the CAMLP4 parser-preprocessor [7], an isolated version of the original OCAML-parser, as a tool for analyzing program text. CAMLP4 generates an abstract syntax tree and

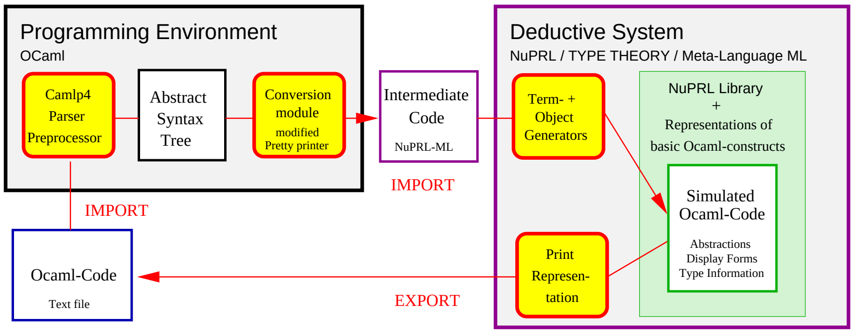

Fig. 2. Translating between OCAML and NUPRL: general methodology

then calls an output module for further processing (e.g. pretty-printing or dumping the binary). We have developed an output module that converts the abstract syntax tree it into "intermediate code" consisting of term and object generators. The module generates pieces of text for each node of the syntax tree, distinguishing the various kinds of identifiers, expressions, patterns, types, signature items, and module expressions according to the OCAML specification [7, Appendix A].

Since a parser is restricted to a syntactical analysis of a single text file it cannot solve problems that arise when linking the code of several modules. *Name resolution* (dealing with modules), determining the *role of identifiers* (variable or reference to user-defined object), and overloading (detecting the intended operator via type inference) therefore had to be addressed with meta-level object generators (see Section 4 of our technical report [14] for details).

Translating formal representations of protocol layers back into OCAML source code is easy, because their display is already genuine OCAML code. Since NUPRL already provides a mechanism for printing libraries and proofs we only had to write a function that selects the objects to be printed. The resulting program text can be executed in the OCAML environment without further modifications.

4 Verification of System Properties

The security of distributed systems is usually described by a few critical system properties such as agreement, total message ordering, safe encryption etc., which are achieved by using specific protocols. Group communication systems built with the ENSEMBLE toolkit consist of stacks of 20–30 small protocols that are composed according to the needs of the application. Since there are thousands of different application systems that can be constructed this way it is not possible to give an *a priori* verification of the ENSEMBLE system as such.

Instead, we have to verify critical properties of *individual* protocol layers and develop proof tactics that derive properties of a full protocol stack from those of individual layers. We also have to state and prove *global* system invariants (e.g. independence of layer properties). These Invariants which must hold for all possible protocol stacks and we have to provide tools that check them whenever the system is modified.

One of the advantages of ENSEMBLE's implementation, as far as formal reasoning is concerned, is the simple code structure of protocol layers. Since the protocols are essentially finite state-event machines, the OCAML code contains only nested function applications, sequencing of statements, and simple loops. Correspondingly, many properties of a protocol layer can be proved by straightforward applications of a few fundamental deductive techniques.

- *Function evaluation* is used whenever a newly defined layer function needs to be analyzed. Similarly, the definitions of newly introduced concepts for expressing program properties have to be unfolded.
- *Lemma application* occurs when reasoning about the properties of operations used in the definition of new functions or concepts. This may involve backward (and forward) chaining over implications and *equality substitutions*, depending on which part of the lemma can be used.
  The only difficulty is finding appropriate lemmata in a short amount of time. Efficient lemma application requires a well-organized *formal database* containing verified lemmata about the properties of predefined functions. A strong modularization of this database is necessary to restrict the search space. We are currently developing techniques for constructing this database step by step as new functions and modules are imported into NUPRL.
- *Proof methods* known from first-order theorem proving are used to solve proof subgoals that are conceptually simple but tedious. Similarly simple induction techniques are needed for reasoning about loops.

Usually the syntactical structure of the property to be proved determines how these techniques are to be applied. This makes it possible to write tactics that automate the verification process. To illustrate this process we give an example verification of the central property of Ensemble's Elect protocol layer.

Example 2. The purpose of the Elect layer is to elect a coordinator for groups of processes that do not suspect each other to have failed. For this, each process maintains a list of suspected processes and its own rank in the group. The former will be updated in each cycle, but processes that have been suspected once will remain suspected. A process elects itself as coordinator if all other processes of lower rank are suspected. This guarantees that *each nonempty subgroup of correct processes will always have exactly one coordinator*. The election algorithm can be implemented by the following piece of OCAML-code.[1]

```
type state = { suspects : bool list ;  rank : int }

let handler (state,suspects) =
  let suspects = map2 (or) suspects state.suspects in
   let elect   = (min_rank suspects) >= state.rank in
    let state  = { rank = state.rank; suspects = suspects } in
       (state,elect)
```

To formalize the property we want to verify, we describe a group G of processes by a list of states. Each member m is uniquely identified by its position, or *rank*, in

[1] map2 applies a function with two arguments to two lists. min_rank computes the smallest rank of an entry false in a boolean list.

this list, which is assumed to be identical with the value of the `rank` component of `G[m]`. The `suspects` component and each list of new suspects are list of boolean values of exactly the same size as the group. They indicate which group member is suspected to have failed. We introduce two formal abbreviations.

$$\texttt{G is\_well\_formed} \;\equiv\; \forall\texttt{m:}\{\texttt{1..|G|}\}.\ \texttt{|G[m].suspects|=|G|} \wedge \texttt{G[m].rank=m}$$
$$\texttt{suspects fits G} \;\equiv\; \forall\texttt{m:}\{\texttt{1..|G|}\}.\ \texttt{|suspects[m]|=|G|}$$

The subgroup `SUB` of correct processes can be characterized as the set of processes that do not suspect each other but suspect every outsider process.

$$\texttt{SUB agrees} \;\equiv\; \forall\texttt{m,m':}\{\texttt{1..|G|}\}.\ \texttt{m} \in \texttt{SUB} \Rightarrow \texttt{m'} \in \texttt{SUB} \Leftrightarrow \texttt{m' unsuspected\_by m}$$
$$\texttt{m' unsuspected\_by m} \;\equiv\; \texttt{suspects[m][m']=false} \wedge \texttt{G[m].suspects[m']=false}$$

A member `m` elects itself as new coordinator, if the `elect`-component computed by its handler is `true`. We define `m elects_itself` as shorthand for

```
let (state,elect) = handler(G[m],suspects[m]) in  elect=true
```

The property that each nonempty subgroup of correct processes will always have exactly one coordinator can thus be formally sspecified as follows.

$$\begin{array}{l}
\forall\texttt{G: state list. G is_well_formed} \Rightarrow \\
\quad \forall\texttt{suspects: } \mathbb{B} \texttt{ list list. suspects fits G} \Rightarrow \\
\quad\quad \forall\texttt{SUB: }\{\texttt{1..|G|}\}\texttt{ list. SUB agrees} \Rightarrow \\
\quad\quad\quad \exists_1\texttt{m:}\{\texttt{1..|G|}\}.\ \texttt{m} \in \texttt{SUB} \wedge \texttt{m elects_itself}
\end{array}$$

A proof of this property decomposes this specification and then continues as shown in Figure 3 (we used descriptions of proof tactics instead of their NuPRL names). Except for the selection of `min(G)` in the first step, it can be constructed automatically by the above-mentioned standard techniques, provided that the following lemmata are present in the formal database.

$$\begin{array}{ll}
\texttt{(1)} & \forall\texttt{L: } \mathbb{B}\texttt{ list.} \quad \texttt{L[min_rank L]=false} \\
\texttt{(2)} & \forall\texttt{L: } \mathbb{B}\texttt{ list.} \quad \forall\texttt{j:}\{\texttt{0..|L|}^{-}\}.\ \texttt{L[j]=false} \Rightarrow \texttt{min_rank L} \leq \texttt{j} \\
\texttt{(3)} & \forall\texttt{L1,L2:}\mathbb{B}\texttt{ list. |L1|=|L2|} \Rightarrow \\
 & \quad \forall\texttt{i:}\{\texttt{0..|L1|}^{-}\}.\ \texttt{(map2 or L1 L2)[i]=false} \Leftrightarrow \texttt{L1[i]=false} \wedge \texttt{L2[i]=false} \\
\texttt{(4)} & \forall\texttt{L:}\mathbb{Z}\texttt{ list.}\forall\texttt{i:}\mathbb{Z}.\ \texttt{i=min(L)} \Leftrightarrow \texttt{i} \in \texttt{L} \wedge (\forall\texttt{j:}\mathbb{Z}.\ \texttt{j} \in \texttt{L} \Rightarrow \texttt{i} \leq \texttt{j})
\end{array}$$

The above example is only an illustration of the tasks involved in the verification of a protocol stack. The actual Ensemble implementation of the Elect protocol layer also contains code for handling various error situations and is about 100 lines long. Because of this, a complete formal specification of protocol layers is rather complex and more difficult to verify. We are currently elaborating formal specifications of Ensemble's protocol layers and the code for layer composition using I/O-automata as in [17]. We intend to use timed automata as a means for specifying synchronization and liveness properties.

The development of verification tactics for protocol stacks is also still in its beginning phase. So far we have developed and implemented a formal programming logic for the embedded subset of Ocaml (see Section 4 of our technical report [14] for details) and experimented with small examples like the above. We have also implemented a type-inference algorithm for the embedded subset of Ocaml and are currently extending it to provide a more detailed analysis of programs like checking array bounds, division errors, boolean annotations, etc.

```
 1.-3. G: state list,        suspects: 𝔹 list list,  SUB: {1..|G|} list
 4.-6.. G is_well_formed,     suspects fits G,         SUB agrees
⊢ ∃₁m:{1..|G|}. m∈SUB ∧ m elects_itself
BY unfold the definition of ∃₁ and choose  m = min(SUB)
|\
|  ⊢ min(SUB) ∈ SUB
|  BY Lemma (4)
|\
| ⊢ min(SUB) elects_itself
| BY unfold elects_itself, evaluate  handler, and convert >=
| |
| ⊢ min_rank (map2 or suspects[min(SUB)]) (G[min(SUB)].suspects)) ≥  G[min(SUB)].rank
| BY unfold is_well_formed and substitute G[min(SUB)].rank=min(SUB)
| |
| ⊢ min_rank (map2  or ... ) ≥ min(SUB)
| BY Lemma (4)
| |
| ⊢ min_rank (map2  or ... ) ∈ SUB
| BY unfold agrees and backward reasoning over hypothesis 6
| |\
| | ⊢ min(SUB) ∈ SUB
| | BY Lemma (4)
| \
|   ⊢ min_rank (map2 or ... ) unsuspected_by min(SUB)
|   BY unfold unsuspected_by and apply Lemma (3)
|   |
|   ⊢ (map2 or ... )[min_rank (map2 or ... )] = false
|   BY Lemma (1)
 \
  7.-9. m:{1..|G|},  m∈SUB,  m elects_itself
  ⊢ m=min(SUB)
  BY Lemma (4)
  |\
  | ⊢ m∈SUB
  | BY hypothesis 8
   \
    10.-11.  j:ℤ,  j∈SUB
    ⊢ m≤j
    BY unfold elects_itself, evaluate handler, and convert >=
    |
    9. min_rank (map2 or (suspects[m]) (G[m].suspects)) ≥ m
    ⊢ m≤j
    BY transitivity over hypothesis 9
    |
    ⊢ min_rank(map or (suspects[m]) (G[m].suspects)) ≤ j
    BY Lemma (2)
    |
    ⊢ (map2 or (suspects[m]) (G[m].suspects))[j] = false
    BY apply Lemma (3) and fold unsuspected_by
    |
    ⊢ j unsuspected_by m
    BY unfold agrees and backward reasoning over hypothesis 6
```

Fig. 3. Top-down verification of the Elect protocol layer

Since the formal specifications of individual layers are not very likely to change, it is sufficient to verify them interactively with tactic support. Formal reasoning about application systems can then be done solely on the basis of these formal specifications instead of the real code, which drastically simplifies the reasoning process. We believe that a formalization of the style of reasoning used in Lynch's book [17] may lead to successful verification tactics.

5 Fast-Track Reconfiguration of Protocol Stacks

For the sake of flexibility, ENSEMBLE's protocol layers can be combined almost arbitrarily. Few assumptions are made about adjacent layers and all types of messages (including errors) must be handled within the layer. This approach is safe but it leads to a great amount of redundancy. In most cases a message is simply sent or broadcast and it passes straight through the protocol stack, modifying only a few layer states. Each layer adds a header to the message to indicate how the corresponding receiver layer has to be activated (see Figure 4).

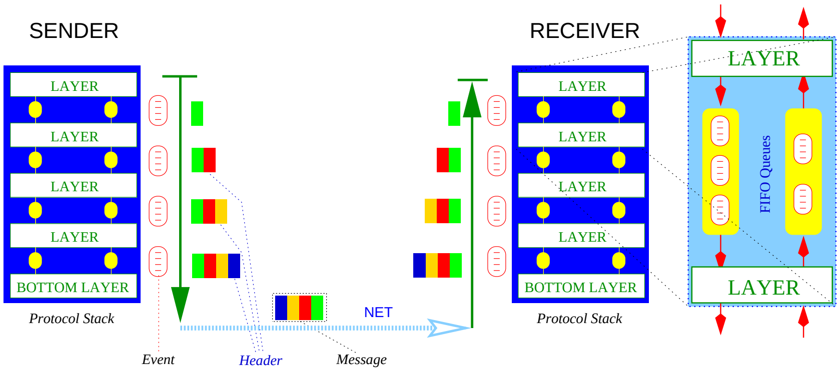

Fig. 4. ENSEMBLE architecture: *Protocol layers are linked by FIFO event queues*

By reconfiguring a layer stack one can achieve a more efficient treatment of these normal cases. For this purpose one has to analyze common sequences of operations, identifying the structure of standard messages and the normal status of a layer's state. Under these conditions the code of the protocol stack can be improved drastically. The result describes a *fast-track* through the protocol stack that can be executed whenever an incoming message and the current state satisfy the conditions. Experiments have shown that a speedup of factor 30–50 can be achieved by function inlining, symbolic code evaluation, dead code elimination, removing the communication overhead between layers, and compressing the headers of standard messages before sending them over the net. Fast-track reconfigurations by hand, however, are time consuming and have a high risk of error because of the code size of typical applications. Without formal support, a reconfiguration of an application system would be infeasible.

We have developed a small set of general tactics that automatically detect pieces of code that can be optimized and rewrite them accordingly. These tactics include function inlining, symbolic evaluation, and knowledge-based simplification. They are based on the derived program evaluation rules mentioned in Section 3 (see also Section 4 of our technical report [14]) and on conditional rewrite rules, which are implemented via substitution and lemma application.

These tactics are successful for a reconfiguration of individual protocol layers. But for the reconfiguration of protocol stacks containing thousands of lines of code they turnead out to be not efficient enough since too much search is involved in the process. Therefore we have developed specialized reconfiguration tactics that avoid search almost completely because by following the code structure of ENSEMBLE's protocol layers and the function for layer composition.

Reconfiguration of protocol layers. Besides type and module declarations all ENSEMBLE protocol layers essentially consists of two functions. The function `init` initializes the layer's state according to a global view state `vs` and local information `ls`. The function `hdlrs` describes how the layer's state is affected when an event is received and which events will be sent out to adjacent layers. Instead of mentioning event queues explicitly, `hdlrs` transforms the event *han-*

dlers of the stacks above and below the layer. This technique makes it possible to `convert` a layer `l` into a functional or an imperative version without modifying its code.

The purpose of a reconfiguration is to optimize the event handler of a protocol stack. A reconfiguration of an individual protocol layer `l` therefore begins with

`let (init,hdlr) = convert l (ls, vs) in  hdlr(`s_l`,`*event*`)`

where `init` describes the initial state, `hdlr` event handler, and *event* is either an up or downgoing event of the form `UpM(ev,msg)` or `DnM(ev,msg)`. Assumptions about the "normal case" characterize the type of `ev` (send, broadcast, etc.), the structure of a header in `msg`, and the contents of the layer's state s_l.

A reconfiguration first evaluates `convert`, which has a fixed implementation, and unfolds `init` and `hdlrs`. It then evaluates all `let`-abstractions occurring in `hdlr` and finally analyzes the outer structure of the event. All these steps could be performed by our general evaluation strategy (a tactic called `Red`) but a specialized tactic `RedLayerStructure` can perform them much more efficiently, because the exact order of reductions is fixed.

Afterwards, we have to make use of the assumptions in order to optimize the code further. Since these are usually expressed as equalities, we can use substitution and then reduce the piece of code that was affected. Again, there is a specialized tactic `UseHyps` for this purpose.

By combining these tactics a reconfiguration of a protocol layer can be performed almost automatically. We have successfully used them in the reconfiguration of several ENSEMBLE layers for various standard situations. 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 that is passed to the next layer.

Verifying a reconfiguration. A fast-track reconfiguration in NUPRL is more than just a syntactical transformation of program code. Since it is based on substitution and evaluation mechanisms we *know* that under the given assumptions a reconfigured program must be equivalent to the original one. But we can also prove this fact formally after a reconfiguration has been finished. In fact, we get the proof of equivalence almost for free, since all NUPRL substitutions and evaluations also correspond to proof rules in NUPRL.

We have written a tactic that generates the statement of the equivalence theorem from the assumptions, the starting point, and the final result of the reconfiguration and then proves it correct. For the latter, it considers the trace of the reconfiguration as a proof plan and transforms each reconfiguration step into the corresponding proof rule and its parameters. This tactic is completely automated – even in cases where the reconfiguration required some user interaction – and is guaranteed to succeed.

An interesting side effect of this technique is that it also allows us to bypass NUPRL's basic inference system *during a reconfiguration*. Since the correctness of the reconfiguration will be verified anyway, we can simply transform OCAML-programs by meta-level operations, which avoids the overhead of checking each transformation step correct. Experiments have shown that this speeds up the reconfiguration process by a factor of 5 and more since the verification can be executed later in a separate background process.

Reconfiguring layer composition. While the reconfiguration of individual layers is mostly straightforward it becomes more complex when layers are composed. In this case the new state is simply a tuple of individual layer states. The composed event handler not only has to deal with outgoing events but also with the events that pass between the composed layers. Since (generated) events may also bounce between the composed layers the function for layer composition (`compose`) must use unrestricted recursion.

Correspondingly, a reconfiguration of composed layers has to proceed recursively as well while it traces the path of standard events through the composed layers. For instance, upgoing events are first be passed to the lower layer. After the code has been reconfigured accordingly, all emitted downgoing events are stored in the event queue while upgoing events will be passed to the upper layer. Emitted up-events are added to the queue and down-events are passed back into the lower layer. This process continues until all events have left the composed layers. It has been automated by a specialized tactic for reconfiguring composed layers, which performs these steps in an efficient order. Obviously, it only 'succeeds' if the assumptions about incoming event do in fact allow a simplification of the code, since otherwise the generated code may become much bigger.

Reconfiguration of protocol stacks. Despite the use of specialized reconfiguration tactics that take into account the specific code structure of ENSEMBLE the performance of the reconfiguration tools does not scale up very well. Due to the size of the code we have to deal with extremely large terms, which is particularly problematic if user interaction is necessary. Tracing events through the code of a full application protocol stack is extremely time consuming as we have to rely on *symbolic* evaluation because we only know the structure of the event.

Reconfiguration in a higher-order proof environment, however, is not restricted to elementary program transformations. In most cases, the result of reconfiguring a protocol stack under a given set of assumptions can easily be derived from the reconfiguration results for its individual layers. We only have to compose these results according to our knowledge about layer composition.

Formally, we can do this by establishing theorems about the reconfiguration of individual protocol layers and the result of composing reconfigured layers. These formal theorems then serve as *derived inference rules* on a much higher level of abstraction. They compose arbitrary protocol layers in a *single* inference step where a tactic-based reconfiguration would have to execute thousands of elementary steps. This not only leads to a better performance of the reconfiguration process but also a much clearer style of reasoning. 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 stack will remain completely unaffected.

Figure 5 presents a reconfiguration theorem for composing fast-tracks for upgoing events. It deals with the very common case of *linear traces* where an event passes through the stack without generating additional events. In this case each layer `Top` and `Bot` yields a queue consisting of a single up-event and the composition of both does the obvious. While the statement of this theorem is

```
∀Top,Bot, ls,vs, msg,msg₁,msg₂, s_b,s'_b, s_t,s'_t
    let (init,hdlr) = Bot (ls,vs) in hdlr (s_b, UpM(ev,msg))  =  (s'_b, [:UpM(ev,msg₁):])
  ∧ let (init,hdlr) = Top (ls,vs) in hdlr (s_t, UpM(ev,msg₁)) =  (s'_t, [:UpM(ev,msg₂):])
⇒    let (init,hdlr) = (compose Top Bot) (ls,vs) in hdlr ((s_b,s_t), UpM(ev,msg))
      = ((s'_b,s'_t), [:UpM(ev,msg₂):])
```

Fig. 5. Reconfiguration theorem for linear up-traces

simple, its proof is rather complex as we have to reason about the actual code of ENSEMBLE's `compose` function and reason about 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.[2]

Reconfiguration theorems for composing fast-tracks, coupled with reconfiguration theorems for individual protocol layers, lead to a reconfiguration technique that scales up extremely well. We only have to apply the appropriate theorems step-by-step and receive the reconfigured code for the complete stack in linear time with respect to the number of layers that have to be passed by events.

We are currently developing a database of standard reconfigurations for all protocol layers, a series of composition theorems for linear and simple non-linear traces, and a reconfiguration tactic that automatically selects and applies these theorems. We have used this tactic in an example reconfiguration of a simple protocol stack consisting of the four layers Bottom, Mnak, Pt2pt, and Frag with total code size of about 1200 lines. Tracing broadcast events under standard conditions yields only *two lines of code*, which update the state of Mnak and pass the event to the next layer after removing the corresponding four headers.

6 Conclusion

We have presented a logical programming environment for the development of reliable and efficient group communication systems. Our approach includes algorithms for importing system code into the NUPRL proof development system, semi-automatic reasoning tools for verifying and optimizing this code within the proof environment, and tools for exporting the results back into the programming environment. It is based on a formalization of a subset of the programming language OCAML for which we have developed a type-theoretical semantics.

Recent work on the specification and verification of timed automata [17, 1], fault-tolerant systems [20], and protocol stacks for group communication systems [8] has demonstrated that formal reasoning about complex distributed algorithms is feasible. Our approach, however, is the first to make the code of a real-world communication system available for formal reasoning and to combine both verification and code reconfiguration within a single formal framework.

While the import/export mechanisms have already been completed, the degree of automation of our tools for verifying and reconfiguring protocol stacks still has to be improved. For this purpose we will integrate additional tools from the field of automated deduction, such as an extended typechecking algorithm

[2] This methodology has already been used successfully for program synthesis [13, 2].

[10], a proof procedure for first-order logic [15], and a proof planner for inductive proofs [2] into the logical programming environment. Furthermore, we are extending our formal database by verified theorems about major reconfiguration and verification steps, which we can then use as derived inference rules. We aim at a modularization of the formal database in order to speed up the search for applicable lemmas. We are also developing a mechanism that automatically adds header compression to a reconfigured stack to further improve the efficiency of the generated code. We intend to apply our reconfiguration and verification tools to a running application system in order to improve its efficiency while hardening its security at the same time.

Although it may take a few years until our tools are mature, we are confident that they will lead to a new design paradigm for distributed systems that yield the high degree of assurance required in many important applications.

References

1. M. Archer & C. Heitmeyer. Mechanical verification of timed automata: A case study. Technical report, Naval Research Laboratory, Washington, DC, 1997.
2. W. Bibel, D. Korn, C. Kreitz, F. Kurucz, J. Otten, S. Schmitt, G. Stolpmann. A multi-level approach to program synthesis. In *Seventh International Workshop on Logic Program Synthesis and Transformation*, LNAI, Springer Verlag, 1998.
3. K. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, 1997.
4. K. Birman & R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.
5. 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.
6. R. Constable, et. al., *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, 1986.
7. D. de Rauglaudre. *Camlp4 version 1.06.* Institut National de Recherche en Informatique et en Automatique, 1997.
8. A. Fekete, N. Lynch, A. Shvartsman. Specifying and using a partitionable group communication service. In *16th ACM Symposium on Principles of Distributed Computing*, 1997.
9. M. Gordon, R. Milner, C. Wadsworth. *Edinburgh LCF: A mechanized Logic of Computation.* LNCS 78, Springer Verlag, 1979.
10. O. Hafızoğulları & C. Kreitz. Dead Code Elimination Through Type Inference. Technical Report TR 98-1698, Cornell University, 1998.
11. M. Hayden. *Ensemble Reference Manual.* Cornell University, 1996.
12. The Ensemble distributed communication system. System distribution and related information. `http://www.cs.cornell.edu/Info/Projects/Ensemble`
13. C. Kreitz. Formal mathematics for verifiably correct program synthesis. *Journal of the IGPL*, 4(1):75–94, 1996.
14. C. Kreitz. Formal reasoning about communication systems I: Embedding ML into type theory. Technical Report TR97-1637, Cornell University, 1997.
15. 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.
16. X. Leroy. *The Objective Caml system release 1.06.* Institut National de Recherche en Informatique et en Automatique, 1997.
17. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.
18. 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.
19. J. Rushby. Formal methods for dependable real-time systems. In *International Symposium on Real-Time Embedded Processing for Space Applications*, pp. 355–366, 1992.
20. 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.
21. R. van Renesse, K. Birman, & S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.