A Language-Based Approach to Protocol Construction

Anindya Basu Mark Hayden Greg Morrisett Thorsten von Eicken

Department of Computer Science, Cornell University, Ithaca, NY 14853-7501

Abstract

User-level network architectures that provide applications with direct access to network hardware have become popular with the emergence of high-speed networking technologies such as ATM and Fast Ethernet. However, experience with user-level network architectures such as U-Net [vEBBV95] has shown that building correct and efficient protocols on such architectures is a challenge. To address this problem, Promela++, an extension of the Promela protocol validation language has been developed. Promela++ allows automatic verification of protocol correctness against programmer specified safety requirements. Furthermore, Promela++ can be compiled to efficient C code. Thus far, the results are encouraging: the C code produced by the Promela++ compiler shows performance comparable to hand-coded versions of a relatively simple protocol.

1 Introduction

Recent research in high-speed network interfaces has focused on removing the operating system from the critical path of communication. An effective solution is to provide *user-level messaging* [vEBBV95, MC95, BDF+95, PLC95, BJM+96], a technique that enables applications to send and receive messages without kernel intervention. The use of user-level networking architectures has pushed in-kernel protocol stacks into user space. As a result, heavyweight, kernel-resident protocols that often add significant overhead to end-to-end latencies can be replaced by streamlined user-level protocols that are tailored for specific applications.

However, experience has shown that writing user-level protocols that are both *correct* and *efficient* is a challenge. For example, race conditions and other errors in a protocol stack are often hard to reproduce for debugging purposes, and even minor changes can break the correctness of a protocol. Furthermore, writing application-specific protocols from scratch for a new application can be very time consuming and error prone.

One of the techniques commonly used to address these issues is to build layered (modular) protocol stacks. Layered protocol stacks divide the protocol into distinct layers, each of which performs some particular function independently of other layers. Simple, individual layers are easier to both write and verify than complex, monolithic stacks and new protocols can be assembled out of existing layers that have been tested and verified. Furthermore, the layered approach facilitates the construction of protocol stacks tailored to application needs. For example, applications that do not require a certain functionality (e.g., encryption/decryption of messages) can simply remove the layer that implements the unneeded functionality, thereby reducing protocol processing overheads. In contrast, it is often difficult to tailor a monolithic stack to specific application requirements without rewriting large portions and introducing errors.

Previous work has demonstrated that layering often adds considerable overheads to protocol processing [CT90]. Thus, the challenge is to combine the modularity and verifiability of layered protocol stacks with the good performance of monolithic (non-layered) protocol stacks. Various partial solutions have been proposed to this problem. Integrated Layer Processing [CT90] advocates the rewriting of existing protocol stacks such that the processing done by the different layers is pipelined, avoiding message copies everytime a layer boundary is crossed. In practice, this is often a complex task and does not allow for code reuse in most cases. The protocol correctness issue has been addressed by multiple high-level validation languages such as Esterel [BG92] and Promela [Hol91]. However, though such languages allow for validation, the set of language abstractions provided by these systems is often very restricted, and support for access to low-level system resources is poor. As a result, specifying efficient protocols using these languages is quite difficult. Moreover, with the exception of HIPPCO [CDO96] (discussed later), compilers that generate efficient code for such languages do not exist. Thus, even if a correct protocol has been developed using these verification tools, the entire protocol must be re-implemented in a different language such as C which may (and often does) introduce errors in the implementation.

This paper presents an overview of the design and initial implementation of Promela++, an extension of the Promela [Hol91] protocol validation language. Promela++ combines the advantages of a protocol validation language with various mechanisms that support efficient code generation from a layered specification. In particular, Promela++ allows the protocol developer to specify the control flow of a protocol layer in a C-like language. This specification can be verified by the Promela validator against programmer-specified safety requirements. For example, the programmer can specify that the protocol never deadlocks by requiring that some particular state in the protocol is reached infinitely often. While such a mechanism does not guarantee absolute correctness, a significant number of errors in the protocol logic can be detected by using a carefully specified set of safety constraints. The Promela++ constructs for efficient composition of multiple protocol layers make it easy for a compiler to perform several optimizations along programmer-annotated "fast paths" that correspond to the more frequently executed code segments in the protocol stack. Finally, Promela++ has a rich set of control and data constructs that are sufficient to specify a wide variety of protocols.

An important benefit of using Promela++ is that the protocol specification need only be written once: the same program can then be converted to Promela for verification, and to efficient C code for execution. The optimizations used by the compiler ensure that the correctness properties are not violated during the Promela++ to C conversion. Compilation of a high-level protocol specification language has been described before [CDO96], but Promela++ appears to be the only language that combines the advantages of protocol validation and optimized code generation from a modular specification.

The rest of the paper is organized as follows. The motivation and a brief description of the language are given in Sections 2 and 3. Section 4 describes the current compiler and a set of compiler optimizations based on the Promela++ language abstractions. An implementation of Active Messages [vECGS92] in Promela++ is described in Section 5 and its performance is evaluated. Finally, Section 6 discusses future work and conclusions.

2 Why a New Language?

In order to significantly aid in the development of complex protocols, a protocol specification language needs to satisfy the following requirements:

- it must allow verification of protocol correctness,
- it must allow efficient code generation from a modular specification, and
- it must have sufficient expressive power to make the specification task reasonably straightforward for a typical protocol programmer.

The following paragraphs discuss the shortcomings of existing languages with respect to these three requirements.

Formal protocol validation languages such as Promela [Hol91] and Esterel [BG92] are well-suited for verifying protocol correctness. However, with the exception of Esterel, there does not appear to be any compiler that generates sufficiently high-performance code for any of these languages. In the case of Esterel, the HIPPCO [CDO96] optimizing compiler generates efficient code from a high-level specification. However, Esterel, like most other protocol specification languages, suffers from a lack of sufficiently expressive language constructs which complicates the programmer's task significantly. For example, Esterel does not support arrays or user-defined types. Furthermore, the syntax and operational model of such high-level languages are often quite different from the widely used C/C++ programming model, making it difficult for protocol programmers to adopt these languages. This is especially true of Esterel, a language that was designed for specifying synchronous digital circuits — a domain that is often quite distinct from the domain of general-purpose, asynchronous network protocols. Finally, network protocols usually need access to low-level operating system services and data structures. Typically, interfaces to system calls are specified in C, and thus, a language whose syntax and semantics differ significantly from C requires extra-lingual support in order to access these services.

From the performance point of view, it appears that C is the language most suitable for writing protocols. Excellent compilers exist for C but it suffers from severe drawbacks as a protocol specification language. First, there is no formal semantic model for C that allows program verification. Second, since C is a general purpose language, it is difficult for a C compiler to make optimizations that are specific to the domain of protocol compilation. There is an ongoing effort to use C with a special compiler that performs certain non-standard optimizations [MPBO96] for the purpose of protocol construction, but this approach does not address the protocol correctness and verification issue.

Finally, no current protocol specification language provides any explicit abstractions for layered protocol stack construction. The Fox project [HL94] has attempted to address this issue using functors and signatures in ML but does not handle protocol verification. Moreover, the ML programming model is very different from that of C/C++ and is quite unfamiliar to most protocol developers.

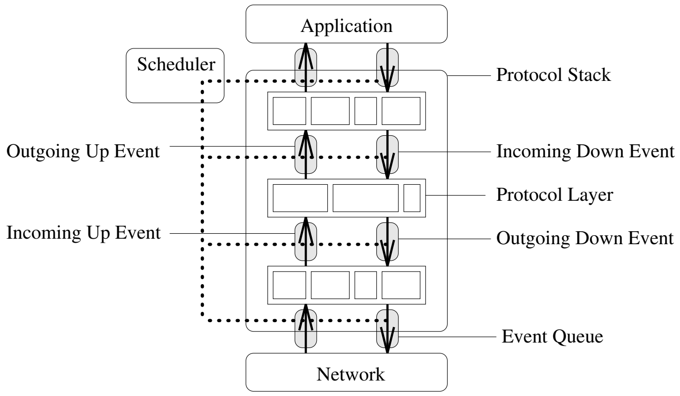

Figure 1: Elements of the execution model.

3 Design of Promela++

Promela++ is an extension of the Promela [Hol91] protocol validation language. The Promela protocol validation language was chosen as a starting point because of its closeness to C and the fact that it was specifically designed for protocol specification and validation. In addition to the features provided by Promela (with some restrictions), Promela++ provides constructs for (a) the layered specification of protocols, (b) the composition of such layers into protocol stacks using an event-based mechanism, (c) the identification of fast paths through programmer annotation, and (d) the encapsulation of protocol state and message headers.

The design goals for Promela++ were the following:

- In order to support automatic verification of correctness requirements, there should be a straightforward translation from Promela++ to Promela. This will allow the use of existing tools, such as the SPIN validator [Hol91], on Promela++ specifications.
- In order to support domain-specific optimizations, Promela++ should have a minimal number of domain-specific language constructs based on a clearly defined execution model.
- The syntax of Promela++ and its expressive power should be as close to C as possible, without compromising the first two goals.

The Promela++ design is based on an execution model that is very similar to the execution model of the Horus system [vRBG+95]. The following subsections first define the Promela++ execution model and then describe how Promela++ directly implements the execution model.

3.1 Execution Model

Conceptually, each protocol layer is represented by a co-routine that communicates with adjacent layers via pairs of asynchronous send and receive channels. More precisely, the layering model consists of the following set of components (see Figure 1):

Event: Events are records used by protocol layers to communicate with other layers. Typically, an event contains a reference to a message that needs to be transmitted or has been received. An event can be an input event or an output event and is associated with a direction that depends on whether the event is moving up or down the protocol stack.

Event queue: Events are passed between layers through directional event queues. Events placed in one end of an event queue are removed from the other end in FIFO order. Each layer is associated with at least four event queues — one each for incoming and outgoing events moving down the protocol stack and one each for incoming and outgoing events moving up the protocol stack. A layer may be associated with more than one outgoing event queue in either direction so that it can implement certain kinds of protocol functionality such as message demultiplexing.

Protocol layer: A protocol layer implements some protocol functionality in the form of an event-based automaton. An instance of a protocol layer consists of a local state, some handlers for processing events passed from adjacent layers, and utility functions that the handlers may call.

Protocol stack: Protocol stacks are created by composing protocol layers. A protocol stack can be visualized as a vertical stack of protocol layers with possibly more than one layer at each level. Adjacent protocol layers communicate through a pair of send/receive event queues. The model also allows non-adjacent layers to communicate directly (using event queues) by bypassing the intermediate layers. This feature helps to make protocol stacks more streamlined for events that an intermediate layer passes on without processing.

Application and Network: The application communicates with the topmost layer of the protocol stack. To send a message, the application directly calls the event handler (in the topmost layer) that is responsible for handling "message send" events. To receive a message, the application registers a callback that is executed by the topmost layer when an incoming message is ready to be passed to the application. Conceptually, this corresponds to the same event queueing mechanism that is used by other layers. However, a function call interface to the protocol stack (as seen by applications) was chosen as opposed to an event-based interface so that applications need not be aware of details regarding event encapsulation and scheduling. Furthermore, a function call interface makes it possible to compile the entire protocol stack into a library that applications can link against.

The physical network is accessible by the bottom layer of the protocol stack which can directly transmit and receive messages over the network.

Scheduler: The scheduler determines the order of event processing in a protocol stack. The scheduler must ensure that events are passed between layers in FIFO order and that any particular layer is processing at most one up and one down event at the same time [1]. This requires that the "up" event handlers manipulate parts of the local state that are disjoint from the parts manipulated by the "down" event handlers so that there are no race conditions. The scheduler must also ensure that every event is eventually processed.

[1] The model allows concurrent processing of one up and one down event so that certain window-based flow control layers may receive acks when they find a filled send window while trying to send a message.

3.2 Promela++: the Language Constructs

The Promela++ language is very similar to C with some modifications and restrictions. In a Promela++ program, the control structure of a protocol layer is separated from the low-level network access and data-manipulation routines. The control structure is specified in Promela++ while the low-level routines are blocks of C code embedded in the program. In essence, the control structure represents a finite state automaton that encapsulates the control flow in the protocol layer. The advantage of such a design is three-fold:

- the design allows the low-level C code to be hidden from the validator which operates only on the control structure for verifying protocol correctness,
- the design allows access to system resources to be smoothly integrated into a protocol specification in Promela++ without the assistance of any external interface mechanism, and
- the entire program needs to be written only once, with the control structure in Promela++ and the low-level C routines as embedded C code. The compiler then generates the appropriate code for verification as well as execution.

3.2.1 States and Messages

In the Promela++ execution model, two of the main components of a protocol layer are the state of the protocol layer and the message header used by the protocol layer. In order to enable the compiler to make certain domain-specific optimizations (described later), the user-defined types used to encapsulate states and messages can be flagged using the keywords `state` and `message`, respectively. A layer can have at most one state type declaration. Lines 1–4 and 9–14 in Figure 2 show sample state and message declarations, respectively.

3.2.2 Inter-layer Communication and Layer Composition

Promela++ provides language constructs that directly implement the event-based inter-layer communication model described in Section 3.1. An event type in Promela++ is a parameterized type that is labelled with a direction that is either *up* or *down* and a qualifier that is either *input* or *output*. The keywords `up`, `down`, `input` and `output` are used to denote these labels. Every input event type must be associated with exactly one event handler that is invoked when the input events of this type are received. When two layers are composed, the input events of one layer are linked to the output events of the other and vice-versa. Note that "down" input events can only be linked to "down" output events and the same holds for "up" events. For inter-layer communication, a protocol layer emits an output event that is linked to the appropriate input event of an adjacent layer [2]. Lines 7–8 in Figure 2 show sample event type declarations while lines 17–35 show a handler declaration. The layers `flow_control` and `network` in Figure 2 can be composed by linking the output event `push` of the `flow_control` layer with the input event `mesg` of the `network` layer as follows:

[2] The output event could be linked to an input event of a non-adjacent layer in case intermediate layers have been bypassed.

```
 message UAM_msg {                /* message type declaration */
     UAM_short   am,
     ....
 };

 layer flow_control;               /* layer declaration */
 /* event type declarations */
 input down req (int dest, handler_t handler, long arg0);
 output down push (int dest, locative UAM_msg mesg, int length);

 state UAM_ctrl {                 /* state type declaration */
     char        win,             /* window size */
     char        mbn,             /* max buffer number (for req and rep) */
     char        high,            /* window high bound */
     ....
};

UAM_ctrl uam_ctrl[MAX_PROCS];     /* local state */

handler uam_request handles req
{
     locative UAM_ctrl ctrl;
     locative UAM_msg msg;
     char diff;

     ctrl = &uam_ctrl[dest];                  /* get pointer to local state */
     msg = &ctrl->req_buf[ctrl->high & ctrl->mbn]; /* ptr to msg to be sent */
     diff = ctrl->high - ctrl->low;
     if
         /* if window not full, go ahead, otherwise wait for acks */
         :: ((diff < 0 ? diff + 128 : diff) != ctrl->win LIKELY) => skip
         :: else => uam_wait (dest)
     fi;
     /* construct message */
     ....
     emit push (dest, msg, 16);           /* pass event to next layer */
     ctrl->high = (ctrl->high + 1) & 127  /* update state */
 }

 layer network;
 input down mesg (int dest, locative UAM_msg msg, int length);
 handler push handles mesg
 {
     /* send on to network */
     C{                                   /* embedded C code */
         int sockfd = lookup_socket (dest);  /* lookup socket */
         /* send using UDP */
         send (sockfd, (char *)msg, sizeof (UAM_msg), 0);
     }C;
     ....                                 /* update state, etc */
 }
```

Figure 2: Sample layers written in Promela++

```
flow_control (push) = network (mesg);
```

The composition is checked by the compiler which verifies that the types of the parameters used by the two events match. The handler for `mesg` is invoked when the output event `push` is emitted (line 33).

3.2.3 Locatives

A characteristic feature of C is the interchangeability of arrays and pointers. C programmers often use pointers to convert array indexing arithmetic to simpler pointer dereferences. The resulting code is usually faster because the addresses of frequently referenced memory locations do not need to be recalculated every time they are accessed. While some C compilers try to eliminate such redundant address computations even when pointers are not used, this optimization is often hampered due to potential aliasing problems when the compiler tries to statically compute the frequently used addresses in order to cache them in registers. Hence, a judicious use of pointers can often lead to significant performance improvements.

Promela does not allow pointers in the language because pointer aliasing can make the task of verification intractable. *Locatives* in Promela++ provide the address caching functionality of pointers in conventional languages, yet are sufficiently restricted so that they can be eliminated entirely at the time of translation to Promela. The Promela++ to Promela translator converts all uses of a locative into references to the array element whose address is stored in the locative. In order to make this task simpler for the compiler, the set of allowable operations on locatives has been restricted in the following manner:

- Locatives may only be initialized and read (i.e., they cannot be updated).
- Locatives must be initialized before they are dereferenced.
- Locatives cannot be initialized in any arm of a conditional.
- Locatives passed as formal arguments to a function must be initialized and cannot be modified in the function body.

Locatives are declared using the keyword `locative` and are assigned and dereferenced exactly as pointers in C. The declaration of locatives is shown in lines 19–20 in Figure 2 and lines 23–25 demonstrate the use of locatives to cache addresses of array elements.

3.2.4 Identification of Fast Paths and Embedded C Code

Promela++ allows programmers to annotate conditionals to indicate which clause contains the commonly executed case and thus should be on the "fast path". Figure 2 shows how the annotations are used. The annotation '`LIKELY`' in line 28 indicates that the condition is true most of the time (this corresponds to the send window not being full in the active message protocol) and the compiler should optimize for this case. If the annotation were '`UNLIKELY`', it would have indicated that the '`else`' case is more frequent and should be optimized. Promela++ also allows programmers to embed C code in the Promela++ specification so that low-level system calls can be integrated into a Promela++ program. In Figure 2, the portion of the code enclosed between '`C{`' and '`}C`' in lines 41–46 (where the message is sent out using the `send` system call) is treated as opaque C code and passed on to the C compiler without processing.

3.3 Verification

The standard Promela distribution [Hol91] includes a verifier called SPIN that does a depth first search of the protocol state space to check for violations of programmer specified safety properties. To use this verifier, the programmer writes a protocol specification in Promela++ along with the safety properties that need to be checked. The safety properties are specified exactly as they are in Promela. The Promela++ compiler can now be used to generate Promela code from the Promela++ specification and the Promela code can be fed to the verifier.

While this approach does not guarantee program correctness, major logical errors in the protocol specification can be detected if a suitably crafted set of safety properties are used for checking the program. For any realistic protocol, the Promela verifier is unable to do an exhaustive search of the entire protocol state space because it is too large and typically, multiple controlled runs of the verifier on disjoint portions of the state space are used to make reasonably accurate assertions regarding protocol correctness.

4 Promela++ to C Compilation

The translation from Promela++ to unoptimized C code is fairly straightforward. The event handlers for each layer constitute a co-routine that asynchronously processes events. The structure of the generated code is shown in Figure 3. Lines 16–17 correspond to an event emission. This code segment marshalls appropriate arguments into an event and enqueues it. The scheduler (called in line 20) services the event queues in round-robin fashion. When it finds a non-empty queue, it dequeues an event off the queue, unmarshalls the arguments and calls the right handler. The scheduler and event management routines are implemented as a protocol-independent runtime library.

4.1 Optimizations

The current version of the Promela++ compiler implements two simple optimizations – first, it converts the event emissions necessary for inter-layer communication into procedure calls wherever possible and second, it inlines procedure calls along the programmer-annotated fast paths (as in [MPBO96]). Converting event emissions to simple procedure calls of the appropriate event handler reduces the layering overheads significantly: the costs of allocating, enqueueing and dequeueing events as well as running the scheduler are now replaced by the cost of a procedure call. For example, the event allocation, enqueueing and scheduling in the unoptimized code segment in Figure 3 are replaced by a call to the handler for the `push` event (line 42).

Inlining along fast paths further reduces layering costs by eliminating procedure call overheads and generating C code that enables further optimizations by the C compiler. These extra optimizations

```
/* handler code in promela++ */
handler uam_request handles req
{
      /* do stuff */
      ....
      /* emit event */
      emit push (dest, handler, arg0);
}


/* generated unoptimized code */
void uam_request (int dest, handler_t handler, long arg0)
{
      Event *ev;
      /* do stuff */
      ....
      /* allocate event, marshall arguments, and enqueue the event */
      ev = __allocate_event (dest, handler, arg0);
      __enq_event (ev, network_down_event_queue);

      /* run the scheduler */
      __scheduler ();
}


/* scheduler code */
void __scheduler (void)
{
     Event *ev;
     /* find the next non-empty queue */
     while (__is_empty (current_queue)) {
          current_queue = next (current_queue);
     }
     /* dequeue entry, unmarshall arguments and schedule event */
     ev = __dequeue (current_queue);
     __call_event_handler (ev);
}


/* generated optimized code */
void uam_request (int dest, handler_t handler, long arg0)
{
      Event *ev;
      /* do stuff */
      ....
      if (__is_empty_and_unlocked_queue (network_down_event_queue)) {
          /* call handler for event 'push' directly */
          network_push (int dest, handler_t handler, long arg0);
      } else {
          /* allocate event, run scheduler, etc */
          ...
      }
}
```

Figure 3: Compiler generated code

are made possible since inlining allows the C compiler (e.g., `gcc`) to optimize across function boundaries using standard techniques such as copy propagation and constant folding.

In order to convert event emissions into procedure calls, the Promela++ compiler needs to ensure that the scheduling constraints are not violated. This involves guaranteeing that:

- a protocol layer does not process more than one up and one down event concurrently,
- the environment in which an event handler executes is identical in both the optimized and unoptimized cases, and
- events are processed in the order in which they are generated (i.e. in the order in which they are generated in the original program without optimizations).

The first requirement is satisfied by using a lock-based mechanism. Every event queue is associated with a lock. The compiler generates code that sets this lock whenever an event popped off the queue is scheduled and resets it in the last statement of the event handler. The generated code now verifies that the appropriate event queue lock is not set and only then calls the event handler as a procedure. Otherwise, the event is enqueued as in the unoptimized case.

The last two requirements necessitate some more analysis. The compiler makes a pass over the entire program and marks event emissions as candidates for conversion to procedure calls if they satisfy the following conditions:

1. the event emission is in the body of a handler, and
2. the event emission is the last statement in the handler body, or is followed only by state updates.

In the unoptimized case, the emitted event's handler executes after the handler that emits the event has completed execution. In the optimized case (where event emissions are converted to procedure calls), this condition may not hold. However, the second condition ensures that the environment in which the event handler for the emitted event executes is identical in both cases (the second requirement) since state updates are local to a layer. Thus, the two versions of the protocol are functionally identical. The last requirement is satisfied because the event emission in question is not followed by any other event emissions.

Note that the first condition requires that the event emission be in the body of a handler for the optimizations to be applied. This can often be restrictive since in many cases, functions called by a handler may emit events. The compiler does apply the same optimization in this case but the corresponding analysis is more complex and has not been described here.

Inlining procedure calls on the fast path is straightforward except that alpha-conversion cannot be performed for variables local to the body of the procedure. This is because the procedure body may have opaque C code that has references to these variables. The compiler simply creates a new block for the procedure body that contains all the local declarations.

Some more aggressive optimizations that include outlining and delayed state updates look promising but have not yet been implemented. *Outlining* [MPBO96] is a technique that improves instruction

Optimization	None	`-O fcall`	`-O inline`	Hand-coded
Latency (in μsec)	176.2	78.8	78.2	77.9
Instruction count (recv)	242	117	112	100

Table 1: Performance of Active Messages

cache performance by compacting the fast path code in one contiguous memory region and moving the infrequently used cases to other regions. Thus, inlining along fast paths and outlining of `UNLIKELY` code fragments results in more efficient placement of code in memory and lower overheads along the fast paths. *Delayed state updates* [HvRB96] improve communication latency by delaying state updates until after the control has been passed to the next protocol layer on the fast path. This results in better overlap of communication with computation and reduces end-to-end message latency. The language constructs of Promela++, including explicit state and event types and programmer-identified fast paths were designed with these optimizations in mind.

5 Implementing Active Messages

The Active Messages communication library has been successfully implemented in Promela++. The Active Messages library is a low-latency communication package for SPMD programs. The computation model for Active Messages assumes that all the communicating processes have identical address spaces. Each active message consists of a handler address and some words of data. On message receipt, the handler extracts the message from the network and incorporates it into the ongoing computation. The Active Messages library supports transmission of small messages as well as bulk transfer of data and has been implemented on MPPs (e.g., CM-5 [vECGS92], SP-2 [CCHvE96], Meiko CS-2 [SS95]) and workstation clusters [vEBBV95].

The Active Messages implementation in Promela++ consists of two layers — a *flow-control* layer that implements a window-based go-back-N retransmission scheme and a *network* layer that communicates directly with the physical network. The microbenchmark that was used measured the round-trip latency for sending a 16 byte active message with a handler that echoes the message. The Promela++ implementation of Active Messages was compiled with two different optimizations and tested against a hand-coded C version. The first optimization (`-O fcall`) converts event emissions into procedure calls wherever possible and the second optimization (`-O inline`) inlines procedure calls along `LIKELY` paths in addition to the first optimization. The compiled as well as the handcoded C versions were all compiled using the `-O2` optimization for the `gcc` compiler. The benchmark platform consisted of two 60 Mhz Sparc-20 workstations with Fore SBA-200 network interface cards running the U-Net firmware and connected to a Fore Systems ASX-200 switch with 140Mbit/sec ATM links.

Table 1 shows the round-trip latencies and number of instructions along the fast paths for each of the implementations described above. The instruction counts are approximate since they were counted by hand and do not include instruction counts for calls to C library functions such as `bcopy`. It is clear that there is a significant improvement in latency (and reduction in number of instructions executed) for the first optimization (`-O fcall`). This can be attributed to the savings in replacing enqueueing, dequeueing and scheduling of events by procedure calls. A comparison with the hand-coded C implementation shows that the optimized versions of the Promela++ implementation perform as well as the hand-coded version.

Note that the performance improvement when procedure calls along the `LIKELY` paths are inlined is negligible. This is primarily due to two reasons: first, the latency microbenchmarks were run on Sparc-20s which use register windows in hardware and consequently have very low procedure call overheads. Second, the Active Messages implementation is rather simple and uses few procedure calls. As a result, the only procedure calls that get inlined are calls to event handlers. In such cases, there is very little overlap between the sets of variables used by the caller and the callee. Thus, the C compiler is unable to further optimize the code using copy propagation and constant folding techniques. The improvement in performance from the `-O inline` optimization should be more significant for complex protocols that have some procedure calls (as opposed to event emissions only) in the `LIKELY` paths.

The Active Messages implementation was also validated against certain simple assertions. The Promela++ to Promela compiler generates about 800 lines of Promela code. The verification was done using the SPIN validator [Hol91] that did a controlled depth-first search of the state space upto a depth of 200,000 nodes and checked about 2 million states for errors. A more exhaustive validation is planned to check for deadlocks and other complex errors.

6 Conclusions and Future Work

Promela++ is a new language that facilitates the development of correct, modular and efficient protocol code. It allows the same source code to be used for verification and efficient code generation. The control flow in the protocol stack is extracted from the Promela++ specification by a converter and fed to the SPIN validator as a regular Promela specification for verification. The code generation process uses a set of optimizations based on special Promela++ language constructs and produces efficient C code. The similarity between C and Promela++ makes it easy to write protocol specifications in Promela++. Finally, a full implementation of the Active Messages library in Promela++ shows performance comparable to that of a hand-coded version.

Experience with Promela++ has revealed two major deficiencies, both of which stem from lack of appropriate language support in Promela. First, Promela++ does not have explicit memory allocation primitives (e.g. `malloc()`). As a result, message allocation and freeing often become cumbersome because programmers must now explicitly implement some form of memory management such as pre-allocating a large chunk of memory out of which messages are allocated at runtime.

Second, Promela++ does not have any support for timers. The only timer related construct in Promela++ (and Promela) is the `timeout` construct which causes an associated "timeout" action to be executed when all other possible actions in the system are blocked. A timeout construct that allows the programmer to specify timeouts in real-time is necessary for a number of protocols such as TCP.

Future work on Promela++ will implement some more aggressive optimizations such as delayed state updates. Inlining procedure calls along the fast path often results in code size explosion because the code in the `UNLIKELY` branches of the called procedure gets inlined too. Ideally, only the code in the `LIKELY` branches of the called procedures should be inlined. Further compiler analysis is required to make this possible. More complex protocols such as the Horus communication system [vRBG+95] are being written, verified and evaluated using Promela++. Finally, profiling information instead of programmer annotations should be used to identify `LIKELY` paths.

Acknowledgments

We are grateful to the anonymous referees for their comments which helped us in improving the presentation significantly.

References

[BDF+95] M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual Memory Mapped Network Interfaces. *IEEE Micro*, 15(1):21–28, February 1995.

[BG92] G. Berry and G. Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[BJM+96] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, 1996.

[CCHvE96] C. C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-latency Communication on the IBM RISC System/6000 SP. In *Proceedings of Supercomputing '96*, 1996.

[CDO96] C. Casteluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. In *Proceedings of SIGCOMM '96*, Stanford University, California, 1996.

[CT90] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of SIGCOMM '90*, pages 200–208, Philadelphia, Pennsylvania, 1990.

[HL94] R. Harper and P. Lee. Advanced Languages for Systems Software: the Fox Project in 1994. Technical Report CMU-CS-FOX-94-01, Carnegie Mellon University, 1994.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[HvRB96] M. Hayden, R. van Renesse, and K. Birman. Optimizing Layered Communication Protocols. Technical report, Cornell University, 1996.

[MC95] A. M. Mainwaring and D. E. Culler. *Active Messages: Organization and Applications Programming Interface*. http://now.CS.Berkeley.EDU/Papers/am-spec.ps, 1995.

[MPBO96] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Latency. In *Proceedings of SIGCOMM '96*, Stanford University, California, 1996.

[PLC95] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, San Diego, California, 1995.

[SS95] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, California, April 1995.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 40–53, Copper Mountain Resort, Colorado, 1995.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[vRBG+95] R. van Renesse, K. P. Birman, B. B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A Flexible Group Communications System. Technical Report TR95-1500, Cornell University, Computer Science Department, March 1995.