A better embedding mechanism would make such analysis possible and thereby ensure smoother and tighter integration of embedded C code. Finally, more exhaustive validations of the Horus and Active Messages protocol implementations need to be done in order to conclusively demonstrate the correctness of these protocol implementations.

References

[ABG96] B. Ahlgren, M. Bjorkman, and P. Gunningberg. Integrated Layer Processing Can Be Hazardous to Your Performance. In *IFIP Workshop on Protocols for High Speed Networks*, Sophia-Antipolis, France, October 1996.

[AP93] M. B. Abbot and L. L. Peterson. A Language Based Approach to Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, February 1993.

[Bas97] A. Basu. *A Language-Based Framework for Protocol Construction*. PhD thesis, Cornell University, August 1997.

[BD95] T. Braun and C. Diot. Protocol Implementation Using Integrated Layer Processing. In *Proceedings of SIGCOMM '95*, Cambridge, Massachusetts, August 1995.

[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.

[Bia94] E. Biagioni. A Structured TCP in Standard ML. In *Proceedings of SIGCOMM '94*, pages 36–45, London, UK, August–September 1994.

[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.

[BMvR96] K. Birman, S. Maffeis, and R. van Renesse. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[Cas97] C. Castelluccia. Personal communication. May 1997.

[CDO96] C. Castelluccia, W. Dabbous, and S. W. 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.

[CWWS92] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is Layering Harmful? *IEEE Network*, 6(1):20–24, January 1992.

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

[HP91] N. C. Hutchinson and L. L. Peterson. The x-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HvRB96] M. Hayden, R. van Renesse, and K. Birman. Optimizing Layered Communication Protocols. Technical Report TR96-1613, Cornell University, Computer Science Department, November 1996.

[Lab92] UNIX System Laboratories. *STREAMS Modules and Drivers, UNIX SVR4.2*. UNIX Press, Prentice Hall, Englewood Cliffs, NJ, 1992.

[LS96] S. Loffler and A. Serhrouchni. Creating Implementations from PROMELA Models. In *Proceedings of the Second SPIN Workshop*, Rutgers University, New Jersey, USA, August 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. W. O'Malley. Analysis of Techniques to Improve Protocol Latency. In *Proceedings of SIGCOMM '96*, Stanford University, California, 1996.

[NO96] E. Najm and F. Olsen. Protocol Verification with Reactive Promela/RSPIN. In *Proceedings of the Second SPIN Workshop*, Rutgers University, New Jersey, USA, August 1996.

[OP92] S. W. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[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.

[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.

[vR96] R. van Renesse. Masking the Overhead of Protocol Layering. In *Proceedings of SIGCOMM '96*, Stanford University, California, 1996.

[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.

Optimization	Instr. Count (Send)	Instr. Cache Miss Rate	Instr. Count (Recv.)	Instr. Cache Miss Rate	Latency μs	Executable Size (bytes)
None	84	0.0061	1259	0.0037	171.2	206856
+ `-O fcall`	84	0.0061	179	0.0086	79.8	207612
+ `-O inline`	84	0.0061	170	0.0087	79.6	208136
+ `-O outline`	85	0.0060	169	0.0087	79.4	208556
+ `-O compress`	85	0.0060	169	0.0087	79.4	208556
+ `-O delay`	85	0.0060	167	0.0088	79.2	208568
hand-coded	50	0.0152	136	0.0083	78.3	218980

Table 3: Performance of Active Messages

Version	BSD	HIPPCO	Promela++
Average Instruction Count	397	193	169

Table 4: Promela++ compared with HIPPCO and BSD

tation of TCP as compared to the BSD and HIPPCO implementations. The HIPPCO and BSD numbers are taken from the literature [CDO96]. The performance figures used for HIPPCO and Promela++ are the ones for the most efficient implementations, i.e., for the implementations that incorporate all the optimizations described in [CDO96] for HIPPCO and all the optimizations described in Section 5.1 for Promela++. The results show that the average instruction count for the Promela++ implementation with all the optimizations is about 12% less than the most efficient HIPPCO implementation and about 57% less than the BSD implementation. Furthermore, the Promela++ implementation consists only of 553 lines of code as compared to 1230 lines of Esterel code, a savings of 55%. A more detailed analysis of the performance figures in order to pinpoint where Promela++ does better than HIPPCO could not be done because we did not have access to the code generated by the HIPPCO compiler.

7 Conclusions and Future Work

Promela++ is a new language that facilitates the development of modular and efficient protocol code whose control structure can be checked for errors using formal methods. It allows the same source code to be used for verification as well as efficient code generation. The control structure in the protocol graph is extracted from the Promela++ specification by a converter and fed to the SPIN validator as a Promela model for verification. The code generation process uses a set of domain-specific optimizations (e.g., conversion of inter-layer communication into procedure calls and delayed state updates) based on special Promela++ language constructs and produces efficient C code. The similarity between C and Promela++ makes it attractive to write protocol specifications in Promela++. So far, use of high-level protocol specification languages has been fairly restricted because of various reasons ranging from unfamiliar programming models to lack of expressiveness and efficient compilers. Promela++ is an attempt to make the use of such high-level protocol specification languages easier for protocol programmers so that they can take advantage of the benefits of these languages, namely verification and domain-specific optimizations. Moreover, implementations of various protocols show that Promela++ does better when compared with hand-coded layered protocol implementations (e.g., Horus) and provides comparable performance when compared with hand-optimized implementations of monolithic protocol implementations (e.g., Active Messages). An important conclusion that can be drawn from this work is that carefully crafted language mechanisms can work together with a protocol compiler to simultaneously provide both ease of programming and efficient code generation. Finally, the Promela++ compiler is also able to generate code that is more efficient than the code generated by an existing protocol compiler, namely, HIPPCO.

Thus far, experience with Promela++ as a language has shown that the domain-specific language constructs in Promela++ (i.e., events, states and messages) clearly expose the underlying execution model. This enables the programmer to reason about protocol correctness without worrying about the low-level details of how the execution model is implemented.

Future work on Promela++ will involve using profiling information instead of programmer annotations to identify fast paths. It has also been observed that certain fields in message headers are unused on the fast paths and can be omitted when sending messages [vR96, HvRB96]. Automatically reducing the size of message headers by generating appropriate code at compile time could have significant impact on the end-to-end latencies of small messages. Further compiler analysis (aided by the `message` data type) is required to make this possible. Furthermore, the embedding mechanism that enables the integration of C code into Promela++ programs does not allow any analysis of such embedded C code by the Promela++ compiler.

Optimization	Instr. Count (Send)	Instr. Cache Miss Rate	Instr. Count (Recv.)	Instr. Cache Miss Rate	Latency μs	Executable Size (bytes)
None	2498	0.0018	2920	0.0017	519.6	201879
`-O fcall`	360	0.0041	663	0.0054	239.5	203796
+ `-O inline`	360	0.0041	636	0.0053	236.4	206254
+ `-O outline`	360	0.0041	635	0.0052	235.9	206686
+ `-O compress`	360	0.0041	637	0.0051	235.1	205844
+ `-O delay`	360	0.0041	635	0.0051	232.5	205796

Table 1: Horus Performance

Version	Horus/C	No Optimizations	All Optimizations
Send Overheads (in μs)	124.0	105.1	38.1
Recv Overheads (in μs)	137.0	139.4	52.5

Table 2: Send and Receive Overheads for Horus

show the cumulative effects of the optimizations in the previous rows. As before, there is a significant improvement in latency (and reduction in number of instructions executed) for the first optimization (`-O fcall`). However, the performance improvement when procedure calls along the fast paths are inlined is much smaller than in the Horus case. This is because the Active Messages code is much simpler and has a lower protocol graph size and no calls to internal functions on the fast paths. The performance figures also indicate that the round-trip times in the optimized case are very close to the round-trip times for the hand-coded version. The other characteristics are similar to the Horus performance except that there is no code size reduction as a result of the `-O compress` optimization. This is because of the nature of the Active Messages code: the Promela++ compiler was unable to find less frequently taken branches in the inlined procedures that could be compressed.

The Active Messages implementation was checked for correctness using the SPIN validator after conversion into a 1300 line Promela model. The validator first checked that the protocol was deadlock-free. Deadlocks occur when the sender and the receiver are blocked waiting for messages from each other and no progress is made. The SPIN validator did an exhaustive search of the state space for all state transition sequences upto 700 steps (approximately 5.4 million states) and no errors were found. The other condition that was checked using the validator was the absence of livelocks. The SPIN validator was again used to check all state transitions of 700 steps or less and no livelocks were detected. It is to be noted that it is possible to run the validator to do an exhaustive search of the entire state space, but the memory requirements for such a process were beyond the capability of the machines at our disposal. The fact that no errors were found in these cases was proof of the fact that the several months of debugging of the original Active Messages implementation (in C) had been effective in removing most of the bugs.

6.3 TCP and Comparison with HIPPCO

A proof-of-concept implementation of TCP was also done in Promela++ in order to compare its performance with respect to the HIPPCO [CDO96] protocol compiler. To do a fair comparison, the Promela++ implementation was derived from the Esterel implementation of TCP described in [CDO96] that was used by the HIPPCO compiler for benchmarking. In order to ensure that the Promela++ implementation supported *exactly* the same functionality as the Esterel implementation in [CDO96], the flow control and error detection algorithms used by TCP were implemented in Promela++, but not the connection setup and teardown algorithms. The Promela++ implementation of TCP consists of two layers — a *network* layer and a *TCP* layer. The network layer converts internet addresses and port numbers into the appropriate endpoint-channel combinations used by U-Net. The TCP layer in Promela++ provides the flow control and error detection functionality. The TCP implementation is described in greater detail in [Bas97].

The TCP implementation was benchmarked using the same application that was used in the HIPPCO case [CDO96]. It consists of a sender that sends a 4 Mbyte file to a receiver that receives the data and sends only pure acknowledgment packets in response (i.e., no piggybacking of acknowledgments is done). The hardware platform used was the same as that in the previous two cases. The HIPPCO benchmark only profiles the control path and does not model message losses — to duplicate this in the Promela++ experiments, instruction counts were measured along the fast paths only and data copies were not performed. The receiver was configured to send one acknowledgment packet for every two data packets received, and the average number of instructions executed to process an input packet was calculated as $(2 * I_c + O_c)/2$ where I_c and O_c are the instruction counts for processing a pure data packet and sending an acknowledgment, respectively. This was the same metric as the one measured in the HIPPCO benchmark.

Table 4 shows the performance of the Promela++ implemen-

first is a subset of the Horus [vRBG+95] communication system for fault tolerant group communication among multiple processes. The second is the Active Messages [vECGS92] communication protocol for low-latency communication in parallel computing. The third is a proof-of-concept implementation of the TCP protocol. The following subsections briefly describe the performance, a more detailed discussion can be found in [Bas97].

6.1 Horus

The Horus [vRBG+95] system consists of multiple protocol layers and provides a framework for the development of distributed applications based on group communications. The Promela++ implementation of the Horus system consists of three of these protocol layers, namely, the *bottom*, *nak* and *stable* layers. The *bottom* layer converts Horus addresses to network addresses and communicates directly with the physical network. The *nak* layer lies above the bottom layer and implements a negative acknowledgment based protocol that ensures reliable and FIFO message delivery. The *stable* layer is the topmost layer and garbage collects messages by reclaiming storage used for these messages after they have been seen by all the members of the current process group. These layers have been discussed in more detail in [Bas97].

A microbenchmark that measured Horus round-trip latencies by ping-ponging 100 byte messages was used to evaluate the performance of Promela++. The benchmark platform consisted of two 60 Mhz Sparc-20 workstations with 20 Kbyte, 5-way set associative level-1 instruction caches and 16 Kbyte, 4-way set associative level-1 data caches. They were connected by a 140Mbit/sec ATM network to a Fore Systems ASX-200 switch using Fore SBA-200 network interface cards running the U-Net [vEBBV95] firmware.

Table 1 shows the round-trip latencies, instruction counts and instruction cache miss rates along the fast paths for each of the optimized versions described above. The numbers in each row correspond to the cumulative application of the optimizations in the previous rows along with the one shown in the current row. It is clear that converting event emissions to procedure calls provides major performance improvements. This is to be expected since the cost of allocating, enqueueing, and dequeueing events as well as running the event dispatcher is replaced by the cost of a simple procedure call. In the case of sends, the other optimizations do not cause any reductions in the number of instructions executed. This is because the send code for the Horus implementation is straight line code without any conditional branches and therefore no inlining or outlining takes place. In the case of message receives, inlining along the fast paths produces a modest decrease in number of instructions executed and also reduces the round trip times. The cache miss ratios in Table 1 reflect only compulsory misses since there are no conflict misses because of the relatively large instruction cache size. This is why outlining produces such a marginal improvement in the instruction cache performance. A different experiment (not described here for lack of space) done with a smaller I-cache size to introduce conflict misses shows that outlining and code compression can improve I-cache performance by a factor of 3.

The Promela++ implementation of the Horus protocol was also compared with the standard Horus implementation in C consisting of the *bottom*, *nak* and *stable* layers. In this case, the software overheads for receiving and sending messages were measured in three cases, namely, (a) the standard Horus implementation in C, (b) the Promela++ implementation compiled without any optimizations and (c) the Promela++ implementation compiled with all the optimizations discussed earlier. Table 2 shows the send and receive overheads in all the three cases. The figures indicate that the performance of the fully optimized version is about 2–3 times better than that of the standard Horus implementation while that of the unoptimized version is comparable to the standard Horus implementation.

The SPIN verifier was used during the protocol construction process to check for protocol correctness and was able to detect two protocol errors in the Promela++ implementation of Horus that were subsequently rectified. The Promela++ program was first converted into a 1000 line Promela model using the Promela++ to Promela translator. Out of the two errors detected in this fashion, one was an incorrect array initialization, and the other was an error in the algorithm that garbage collects messages.

6.2 Active Messages

The Active Messages [vECGS92] library is a low-latency communication package for SPMD programs. 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 at the recipient. The Active Messages library provides primitives for low-latency sends of small messages (handler plus up to 4 words of data) as well as high-bandwidth bulk transfers such as stores and gets.

The Active Messages implementation in Promela++ consists of three layers. The *network* layer communicates directly with the physical network and demultiplexes received messages between the *request_flow_control* layer and the *reply_flow_control* layer. These two layers together implement a window based flow control strategy with go-back-N retransmissions. The flow control protocol for Active Messages has been described in more detail in [Bas97].

The microbenchmark used for evaluating the Active Messages performance measured the round-trip latency for sending a 16 byte active message with a handler that echoes the message. The optimizations and the hardware platform used in this case were the same as described in Section 6.1. In addition, the various optimized versions were compared against a hand-coded monolithic C implementation of the same protocol that was run over the same ATM network using U-Net.

Table 3 shows the performance of Active Messages on the fast paths with respect to instruction counts, cache miss rates and round-trip latencies. Again, the numbers in each row

mizations (described later), the user-defined types used to encapsulate states and messages are flagged using the keywords `state` and `message`, respectively. The event-based inter-layer communication model described previously is directly implemented by the *event* type which is a parameterized type that is labeled with a direction that is either *up* or *down* and a qualifier that is either *input* or *output*. Every input event type must be associated with exactly one event handler that is invoked when 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.

Other features of Promela++ include *locatives* that are a restricted form of pointers that provide the address caching functionality of pointers in general purpose languages without making the task of verification intractable due to aliasing problems, and deterministic `if` statements that have the same semantics as `if-then-else` statements in C. In order to support integration of low-level system calls, Promela++ programs allow C code to be embedded in them using the keywords '`C{`' and '`}C`' as delimiters.

4.1 Limitations

The Promela++ language has two major limitations, both of which stem from the lack of appropriate language support in Promela. First, Promela++ does not have explicit memory allocation primitives (e.g. `malloc()`) which makes message allocation operations difficult. 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. Without a timeout construct that allows the programmer to specify timeouts in real-time, implementation of a number of widely-used protocols such as TCP becomes quite cumbersome, if not impossible.

4.2 Verification

The Promela++ framework supports protocol verification by providing a Promela++ to Promela translator that converts Promela++ specifications to Promela models. It should be noted that only the control structure of the Promela++ specification is converted to Promela: the embedded C code is not translated to Promela and therefore not checked by the verifier. The SPIN protocol verifier [Hol91] (that is part of the standard Promela distribution) is used to convert the Promela model into a state based automaton and then do a depth-first search of the state space to check for violations of user-defined correctness conditions. However, the verification process is only a tool to detect hard-to-identify logical errors in the program and does not guarantee program correctness. Furthermore, although the Promela++ to C compiler does attempt to prevent the introduction of errors during the compilation process by maintaining certain invariants (described in detail in [Bas97]), a formal guarantee of the correctness of the generated code can only be offered if the Promela++ compiler itself is verified, which is beyond the scope of this work.

5 Promela++ to C Compilation

The translation from Promela++ to unoptimized C code occurs as follows. The generated code for an event emission marshals the appropriate arguments into an event and enqueues it. The event dispatcher services the event queues in round-robin fashion by dequeueing events off non-empty queues and calling the associated event handlers. The dispatcher and event management routines are implemented as a protocol-independent runtime library which is single-threaded although the execution model does not specifically preclude a multi-threaded implementation. However, some of the compiler optimizations (described in the next subsection) may need to be rewritten for a multi-threaded implementation.

Although the dispatcher services the event queues in round-robin order, it should be noted that the architecture does not preclude other scheduling models such as assigning priorities to event queues. However, this requires a rewriting of the dispatcher function in the runtime library. At present, only the round-robin policy is supported.

5.1 Optimizations

The current version of the Promela++ compiler implements five optimizations, namely: (a) the `-O fcall` optimization converts the event emissions necessary for inter-layer communication into procedure calls — this reduces the layering overheads significantly by eliminating the costs of allocating, enqueueing and dequeueing events as well as running the event dispatcher, (b) the `-O inline` optimization inlines procedure calls along the programmer-annotated fast paths, (c) the `-O outline` optimization [MPBO96] improves instruction cache performance by compacting the fast path code in one contiguous memory region and moving the infrequently used cases to other regions, (d) the `-O compress` optimization prevents code size explosion by only inlining the code along the fast paths of inlined procedure calls instead of the entire procedure, and (e) the `-O delay` optimization delays state updates [vR96, HvRB96] 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.

6 Protocol Implementation and Performance

In order to evaluate the effectiveness of Promela++, three different protocols have been implemented in Promela++. The

Again, Promela++ addresses these issues by providing explicit language constructs that encapsulate layering semantics and a clean execution model for inter-layer communication. Finally, the code generated by the HIPPCO compiler cannot be run as-is on a real network [Cas97]. On the other hand, the Promela++ compiler generates code that can be run on real computer networks without any modifications. The next section provides a brief introduction to Promela in order to motivate the need for Promela++.

3 A Brief Introduction to Promela

The Promela [Hol91] protocol validation language was used as a starting point for the design of Promela++ because of its closeness to C and the availability of the SPIN model checker for verification of Promela programs. A Promela program models the control structure of a program which can be checked against a set of user-defined correctness conditions using SPIN. The three main object types in Promela are *processes*, *channels* and *variables*. Processes model state machines that communicate with each other by asynchronous message passing across FIFO channels while variables store process-specific or global state. Control flow in Promela is provided by the `if`, `do` and `goto` constructs. The `if` construct is non-deterministic which means that if multiple conditional guards evaluate to true, only one conditional is non-deterministically chosen and executed. The `do` construct enables repeated execution of statements and the `goto` construct provides unconditional jumps. *Atomic sequences* model execution of statement sequences in a process without interleaving them with the execution of statement sequences from other processes. *Timeout* statements in Promela can be used to model timeouts in real systems. Promela also provides language constructs for specifying various correctness conditions such as absence of deadlocks, absence of livelocks and so on. A detailed description of these can be found in [Hol91].

Although, Promela is well-suited for checking protocol specifications for logical errors, it does not support many programming idioms that are commonly used in general purpose programming languages, e.g., procedure calls, pointers and so on. Moreover, non-deterministic `if` statements make the semantics of Promela very different from the semantics of deterministic general-purpose programming languages. In addition, Promela has no explicit layering constructs that makes the task of modular construction difficult. An extension to Promela, called Reactive Promela [NO96], does provide modularity by allowing independent finite state automata to communicate with each other through well defined input/output interfaces. However, inter-automaton communication is modeled as synchronous communication between reactive systems which is different from the inherently asynchronous inter-layer communication of network protocols. Finally, no *efficient* Promela compiler has been described in the literature — work by Löffler et. al. [LS96] describes a Promela to C compiler, but there is no data on how efficient the generated code is.

4 Design of Promela++

Promela++ was designed to address the issues described in the previous section. In addition to the language constructs provided by Promela (with some restrictions), Promela++ provides constructs for (a) the layered specification of protocols, (b) the composition of such layers into protocol implementations using an event-based mechanism, (c) the identification of fast (or, commonly used) paths through programmer annotation, and (d) the encapsulation of protocol state and message headers.

The Promela++ design is based on an execution model that is derived from the Horus runtime system [vRBG+95, BMvR96]. The following paragraphs briefly describe the Promela++ execution model and the Promela++ language constructs (a more detailed description can be found in [Bas97]).

A protocol implementation in Promela++ can be visualized as a graph (as in the x-kernel [HP91]) where the nodes represent protocol layers and the edges represent inter-layer communication. Conceptually, each protocol layer is a co-routine that communicates with adjacent layers via *events* that can either be *down* or *up* depending on whether the event is moving towards the network from the application or away from the network to the application. Incoming events for a layer are processed in FIFO order. Each layer has at least four event queues: one each for incoming and outgoing down events and one each for incoming and outgoing up events. An event dispatcher ensures 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 any given time. It also makes sure that every event is eventually processed.

The Promela++ execution model is very similar to the Unix STREAMS [Lab92] model where the modules correspond to protocol layers, the streams correspond to protocol graphs and the upstream and downstream events correspond to up and down events, respectively. A major difference is that in the case of Promela++, every protocol graph has its own event dispatcher and (potentially, its own) scheduling policy provided by the runtime library (Section 5), whereas in the case of STREAMS, there is only one system-wide run queue where events belonging to all streams are enqueued and then scheduled in FIFO order.

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 and communicates with the external world using the low-level C routines.

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 opti-

tions in Promela++. The evaluation consists of validating (automatically generated) Promela models of the protocols with respect to user-specified correctness conditions, as well as generating optimized code from these specifications. The performance of the generated protocol code on an ATM network is shown to be better than or comparable to that of the corresponding hand-optimized protocols.

The rest of the paper is organized as follows. Section 2 discusses related work and motivation. The Promela language is briefly discussed in Section 3 followed by a description of Promela++ in Section 4. Section 5 describes the current compiler and a set of compiler optimizations based on the Promela++ language abstractions. The performance of Promela++ implementations of multiple protocols is evaluated in Section 6. Finally, Section 7 discusses future work and conclusions.

2 Motivation and Related Work

Techniques for construction of modular and efficient protocol implementations have been studied extensively. The Integrated Layer Processing (ILP) [CT90] technique advocates implementing protocols in a way that pipelines the processing done by individual layers – this eliminates message copies when layer boundaries are crossed. However, later work has shown that the performance improvements due to ILP are extremely sensitive to the nature of the data manipulation operations [BD95] and may not provide expected performance gains in some cases [ABG96, BD95]. The microprotocol approach [OP92] argues in favor of splitting existing coarse-grained and complex protocol layers into simpler, fine-grained microprotocol objects to be combined at runtime, and shows that improved performance is possible even when a fairly large number of such microprotocols are combined. The Morpheus [AP93] language is an attempt to enforce the microprotocol design philosophy using language constructs — however, no compiler for automatically generating executable code from a Morpheus specification has been described in the literature. Other work has focused on circumventing layering overheads by identifying and applying a set of optimizations specific to the domain of network protocol construction [MPBO96, vR96]. While these optimizations have produced significant improvement in protocol performance, the programmer's task still remains complicated because the optimizations have to be applied manually.

The approach described in this paper is language-based and differs from the work described above in that it

- uses a *specification language* to take advantage of existing protocol validation tools,
- exploits *domain-specific information* (from the specification language) for optimizations that may not be possible in a general purpose language,
- generates executable code *automatically* from a high-level specification to which formal protocol validation techniques can be applied for detecting errors in the control structure, and
- offers *modular specification* of protocol layers that can be composed at compile time for the desired functionality.

The idea of using a high-level language for protocol specification has been used before in protocol validation languages such as Promela [Hol91] and Esterel [BG92]. These languages are well-suited for verifying protocol correctness. However, as explained in Section 1, protocol validation languages are hard to use because they often lack support for commonly used programming idioms such as procedure calls, records/structs, pointers, and arrays. In addition, most current protocol specification languages do not provide explicit abstractions for layered protocol construction. The FoxNet project [Bia94] has attempted to address this issue using functors and signatures in ML but does not handle protocol verification. Moreover, latencies for the FoxNet TCP implementation over Ethernet are an order of magnitude larger compared to latencies over standard C implementations, while bandwidths are about four times worse [Bia94].

The Promela++ approach addresses these issues by providing:

- a rich set of language constructs that allow programmers to easily specify a wide variety of protocols,
- explicit language mechanisms to encapsulate protocol layering and efficient composition of such layers, and
- closeness to the C/C++ programming model that makes it easier and attractive for protocol programmers to use Promela++.

In the domain of protocol compilation, the HIPPCO [CDO96] protocol compiler generates very efficient code from high-level protocol specifications in Esterel. The Promela++ compiler (that is part of the Promela++ framework) is similar in spirit to the HIPPCO compiler and indeed draws inspiration from that work for some of its optimizations. However, the HIPPCO approach has some drawbacks. First, it uses the protocol specification language Esterel that has several disadvantages which have been described in the previous paragraphs. Second, it is not clear how protocol validation techniques can be integrated into the HIPPCO framework. Although the authors point out that infrastructures for formal validation of Esterel specifications exist, they do not actually attempt to validate the TCP implementation discussed in their work. On the other hand, the Promela++ framework provides a translator to Promela such that the SPIN verification tool can be used and demonstrates the effectiveness of this approach by actually validating two different protocol implementations in Promela++. Third, there is no explicit support for layering, modular protocol construction or protocol component reuse in either HIPPCO or Esterel.

Promela++: A Language for Constructing Correct and Efficient Protocols

Anindya Basu
Bell Laboratories
basu@research.bell-labs.com

Greg Morrisett
Cornell University
jgm@cs.cornell.edu

Thorsten von Eicken
Cornell University
tve@cs.cornell.edu

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. As a consequence, heavyweight, kernel-resident protocols can be replaced by streamlined user-level protocols that are tailored for specific applications. Flexibility and protocol reuse become important issues in this context because different application domains have widely varying requirements and developing new protocols for every new application domain can be a time consuming task. Furthermore, debugging and optimizing network protocols is a complex process. Thus, in order for user-level messaging to enable the wide-spread use of application-specific protocols, the difficulties of the protocol-development task must be addressed.

High-level protocol specification languages such as Promela [Hol91] and Esterel [BG92] have often been used to check protocols for correctness. However, most of these languages suffer from a lack of sufficiently expressive language constructs and have programming models that are very different from the familiar C/C++ programming models. As a result, programmers find it difficult to adopt and use such languages. Flexibility and protocol reuse issues have traditionally been addressed by layering techniques where each protocol is decomposed into multiple independent units called layers that are then combined to provide the complex protocol functionality. Simple, individual layers are easier to both write and debug than complex, monolithic implementations. Moreover, new protocols can be composed out of existing layers that have been previously tested and verified, thereby allowing protocol reuse. However, previous work has demonstrated that layering (and more generally, modularity) may add considerable overheads to protocol processing [CT90, CWWS92]. Techniques for circumventing layering overheads have used various methods such as ILP [CT90] and microprotocols [OP92] that are not always easy to use or apply by hand. Therefore, the challenge is to develop an easily usable protocol development framework that combines the flexibility of layered implementations, the efficiency of tightly-coupled monolithic implementations and the correctness achievable using high-level protocol validation languages.

This challenge is addressed by a language-based framework that introduces a new protocol specification language called Promela++. The framework consists of a protocol verification tool and an optimizing compiler that generates efficient protocol code from Promela++ specifications. Promela++ is based on the Promela [Hol91] protocol validation language and has been designed with a rich set of domain-specific[1] constructs. These constructs facilitate the task of protocol specification as well as enable the Promela++ compiler to perform domain-specific optimizations. The Promela++ compiler can also automatically transform protocol specifications in Promela++ to protocol models in Promela. The SPIN validator [Hol91] can then be used to check the Promela models against programmer specified correctness conditions for errors in the control logic of the protocol. It should be noted here that the model checking is performed only on the control structure of the protocol and is not meant to detect general programming errors such as dereferencing null pointers.

An important benefit of using Promela++ is that the protocol specification need be written only once: the same specification can then be converted to "pure" Promela for verification, and to efficient C code for execution. The process of compiling Promela++ to C maintains certain invariants in order to prevent errors from being introduced during compilation. However, formal guarantees about the correctness of the generated code cannot be made since that would require a formal verification of the compiler itself which is beyond the scope of this work. The purpose of protocol verification is to discover hard-to-detect logical errors in the protocol, such as those caused by race conditions, and *not* to generate provably correct code.

The main contribution of this work is three-fold:

- it presents a new language that unites the twin goals of checking protocol correctness using model checkers, and efficient protocol construction using optimizing compilers, under a single framework,
- it exploits language design to provide mechanisms that simultaneously ease programming and enable generation of efficient protocol code, and
- it demonstrates the effectiveness of this approach by doing a complete evaluation of multiple protocol implementa-

[1] The term "domain-specific" means that the language constructs are specific to the domain of network protocol construction.