Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols

Satish Chandra, Member, IEEE, Bradley Richards, and James R. Larus, Member, IEEE

Abstract—In this paper, we describe Teapot, a domain-specific language for writing cache coherence protocols. Cache coherence is of concern when parallel and distributed systems make local replicas of shared data to improve scalability and performance. In both distributed shared memory systems and distributed file systems, a coherence protocol maintains agreement among the replicated copies as the underlying data are modified by programs running on the system. Cache coherence protocols are notoriously difficult to implement, debug, and maintain. Moreover, protocols are not off-the-shelf, reusable components, because their details depend on the requirements of the system under consideration. The complexity of engineering coherence protocols can discourage users from experimenting with new, potentially more efficient protocols. We have designed and implemented Teapot, a domain-specific language that attempts to address this complexity. Teapot’s language constructs, such as a state-centric control structure and continuations, are better suited to expressing protocol code than those of a typical systems programming language. Teapot also facilitates automatic verification of protocols, so hard to find protocol bugs, such as deadlocks, can be detected and fixed before encountering them on an actual execution. We describe the design rationale of Teapot, present an empirical evaluation of the language using two case studies, and relate the lessons that we learned in building a domain-specific language for systems programming.

Index Terms—Domain-specific languages, distributed systems, cache coherence, continuations, verification.

1 INTRODUCTION

Distributed systems often cache, or make local copies of, nonlocal data in order to achieve higher performance or availability (or both). As the nonlocal data undergoes modifications, the cached copies must be kept consistent with these modifications; otherwise distinct nodes in the system may observe divergent values for the same data. An algorithm that manages the consistency of the cached copies is called a cache coherence protocol.

Cache coherence protocols are key components in many parallel and distributed systems. Distributed shared memory systems [1], [2] implement a system-wide shared address space over a network of workstations by automatically caching nonlocal data into local memory. These cached copies must be kept consistent with modifications to shared data being made by an application program running on the system.1 Distributed file systems [3], [4], which allow files to be shared over a network, usually employ caching and implement a cache coherence protocol. High-performance client-server database systems [5] also implement cache coherence protocols. Coherence in web caching is a current research topic in the distributed systems community [6].

1. Readers can find background material on the implementation of distributed shared memory, as well as details of a sample cache coherence protocol in Section 2.

Tools that facilitate the implementation of cache coherence protocols are important for two reasons. First, coherence protocols, while common, show a great deal of variety because the protocol for a particular system is closely linked to its sharing semantics and performance goals. For example, different distributed shared memory systems provide different memory consistency models [7]. Moreover, systems with similar sharing semantics can have vastly different protocols that use different algorithms to achieve the same task, albeit with different efficiency. Thus, each system essentially needs its own coherence protocol.

Second, and perhaps more importantly, cache coherence protocols represent complex, distributed algorithms that are difficult to reason about, and often contain subtle race conditions that are difficult to find through system testing. Furthermore, to our knowledge, previous systems have not attempted a clear separation between the cache-coherence engine and other implementation details of the system, such as fault management, low-level I/O, threads, synchronization, and network communication. It is not difficult to imagine the hazards of this approach. The implementor cannot reason about the coherence protocol in isolation from other details, and any modification she makes in the system can potentially impact the protocol’s correctness—a debugging nightmare. Experimentation with newer protocols is a perilous proposition at best.

1.1 Teapot

Teapot is a protocol writing environment that offers two significant improvements over the previous approach of writing ad hoc C code. First, it is a domain-specific language specifically targeted at writing coherence protocols. As such, it forces a protocol programmer to think about the logical structure of a protocol, independent of the other parts of a system. The language features of Teapot...
easily express the control structures commonly found in coherence protocols. Second, Teapot facilitates automatic verification of protocols because it not only translates Teapot protocols into executable C code, but also generates input code for MurΦ, an automatic verification system from Stanford [8]. MurΦ can then be used to detect violations of invariants with a modest amount of verification time. For example, our system can report a sequence of events that would cause a deadlock. A protocol can be run through a verification system prior to actual execution to detect possible error cases, without having to manually rewrite the protocol in MurΦ’s input language.

This paper describes the Teapot system, and our (and others’) experiences with using Teapot to implement the coherence engines in two distinct systems. The first system, loosely coherent memory (LCM) [9], implements a particular type of distributed shared memory suitable for data-parallel programming. The second system, the xFS distributed file system [10], implements a distributed, high-performance, serverless file system. Implementors of both systems found Teapot to be vastly superior to earlier efforts to implement the protocols using C without domain-specific tools. In particular, we found the following benefits when using Teapot:

- **Domain-specific language constructs**: Teapot’s state-centric control structure and continuations considerably simplified the task of writing protocol code, compared to expressing the same code in C.
- **Modularity**: Protocol code in Teapot was more modular, because Teapot forced implementors to separate the protocol logic from the low-level details of the system.
- **Debugging**: Automatic protocol verification using the MurΦ system improved system confidence and reduced testing time.

Perhaps more importantly, this paper also discusses shortcomings of the language that became apparent only when we attempted to develop protocols that were more complicated than the simple protocol examples on which Teapot was originally tested. In particular, our experience indicates that improved support for multithreaded environments, for protocol actions that affect multiple cache blocks, for local protocol actions that might wait on an event, and for automated verification test strategies would further ease the job of a protocol designer.

Finally, the paper generalizes our experience to provide guidelines for future domain-specific languages for systems software.

This paper is organized as follows. Section 2 provides background material on cache coherence protocols and describes the implementation problems faced by protocol programmers. Section 3 presents Teapot, emphasizing the features that address the difficulties presented in Section 2. Section 4 presents an empirical evaluation of Teapot by detailed case studies on using it to write protocols for LCM and xFS. We compare various aspects of our work with related work in Section 5. Section 6 concludes the paper with implications for domain-specific languages for systems software.

2. Continuations are described in Section 3.1.
home node for a readable copy and await a response. Assuming no outstanding writable copy exists (the \textit{Idle} state in Fig. 1), the home responds with a readable copy and changes its state to \textit{ReadShared}. The arrival of this message on the nonhome side causes the protocol to copy the incoming data to memory and change the block’s state to \textit{Readable} (and access permissions are changed from \textit{invalid} to \textit{readonly}).

Unfortunately, specifying protocols is much more difficult than the simple three-state diagrams in Fig. 1 would lead one to believe. The main difficulty is that, although the transitions shown appear to be atomic, many state changes in response to protocol events cannot be performed atomically. Consider the transition from the \textit{Exclusive} state to the \textit{ReadShared} state in Fig. 1. Conceptually, when a request arrives in the \textit{Exclusive} state for a readable copy of a block, the protocol must retrieve the exclusive copy from the previous owner and pass it along to the requestor. The protocol sends an invalidation request to the current block holder, and must await a response before proceeding (Fig. 2).

However, the protocol processing resource (typically, a coprocessor) is shared by a large number of cache blocks, and handlers must quickly run to completion and relinquish the processor to avoid deadlock in a real system. Handlers cannot wait (or \textit{block}) on an asynchronous event, such as a message arrival. This requires that an intermediate state, \textit{Excl-To-ReadShared} (\textit{Excl-RS} for short), be introduced (Fig. 3). After sending the invalidation request, the protocol moves to the \textit{Excl-RS} state and relinquishes the processor. When the invalidation acknowledgment arrives in this intermediate state, the processor sends a response to the original requestor and completes the transition to \textit{ReadShared}. A revised state diagram incorporating the required intermediate states is shown in Fig. 4 (which is still far removed from a realistic protocol).

Introducing intermediate states increases the number of states a programmer has to think about. Furthermore, while in an intermediate state, messages other than the expected reply can arrive. For example, before the invalidation response arrives in the \textit{Excl-RS} state, another request for an exclusive copy could arrive from a different processor. A protocol designer must anticipate the arrival of such unsolicited messages and handle them in an appropriate manner. It may be tempting to not take such messages out of the network while they are not welcome. This, however, is not an option on most systems, because messages must constantly be drained from the network to avoid deadlock in the network fabric [11].

<table>
<thead>
<tr>
<th>State</th>
<th>Exclusive</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ReadRequest(Node R)</td>
<td>Send(Owner, PutRequest);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sharer := R;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>State := Excl-To-ReadShared;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exit;</td>
</tr>
</tbody>
</table>

Fig. 2. Action code when a block receives a \textit{ReadRequest} message in state \textit{Exclusive}. This action code uses a blocking \textit{AWAIT}, which could cause deadlock in a real system. (This code is written in pseudo-Teapot: \textit{AWAIT} is not available in the language, and some syntactic detail has been omitted.)

<table>
<thead>
<tr>
<th>State</th>
<th>Excl-To-ReadShared</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PutResponse(Node R)</td>
<td>SendSharedCopy(Sharer);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>State := ReadShared;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exit;</td>
</tr>
</tbody>
</table>

Fig. 3. This action code shows a way to avoid synchronous communication inside a handler. The intermediate state, \textit{Excl-To-ReadShared}, is used to wait for the reply message. Both handlers can now run to completion without waiting for network messages.
Message reordering in the network adds to the woes of a protocol programmer. For example, processors may appear to request copies of cache blocks that they already have, if a read request message overtakes an invalidation acknowledgment message in the network. The protocol might have to await delayed messages before deciphering the situation and determining the correct action. Without machine assistance, anticipating and properly handling all possible network reorderings is a very difficult task!

The traditional method of programming coherence state machines usually resorts to ad hoc techniques: unexpected messages may be queued, they may be negatively acknowledgment (nack’ed), or their presence may be marked by a “flag” variable. Additional flag variables are often used to track the out-of-order arrival of messages as well. These techniques invite protocol bugs. Queuing can easily lead to deadlocks; similarly, nack’ing can lead to livelocks or deadlocks. Flag variables are essentially extra protocol state—failing to update or test a flag at all the right places again leads to correctness problems.

Protocols implemented in this style are also difficult to understand and modify, even though the preferable model of development of a new protocol is to modify a closely resembling existing one; protocols are seldom written from scratch. Consider adding a *Compare&Swap* primitive to the basic protocol. This primitive is a minor variation of a *WriteRequest* that also executes the atomic compare and swap operation at a block’s home node once the block becomes *Idle*. In our case studies, tracking a pending *Compare&Swap* complicated nearly every transition in a home node state machine. In one state machine-based implementation, this modification manifested itself at 14 different places. Every one of such cases would have an impact on either correctness or performance, or both.

3 **Teapot**

Teapot is a protocol writing environment that includes a domain-specific language and support for automatic verification. The Teapot language resembles Pascal with extensions for protocol programming support, but with fewer built-in types. The Teapot compiler can generate executable C code from a protocol specification, and can also translate it to code that can be given to the Murϕ verification system [8]. A distinctive feature of Teapot as a language for expressing protocol is to structure the control flow of a protocol using continuations rather than a flat state machine. We first motivate the use of continuations in writing protocols, and then proceed to describe Teapot’s language features and verification support.

3.1 Continuations and Protocols

Recall from Section 2 that the primary difficulty in writing protocols as state machines is the need to encode all control information explicitly in states. In this section, we show that a control mechanism used in functional languages, continuations, allows a natural expression of the control flow in coherence protocols.

A continuation captures the execution state of the current computation, stops it, and passes the encapsulated state as an object to another computation. The second computation can resume the captured computation when and if it so desires. The difference between a procedure called via a continuation and an ordinary procedure call is that in the case of continuations, the evaluation of the called procedure logically happens in a separate execution context (or, stack). This language feature enables multiple execution contexts to coexist, and has been used to provide multiple threads [12]. While we could address the problem of blocking communication by handling the protocol processing for each block in a separate, system-implemented thread, continuations provide several additional advantages, which will become apparent in this section.

Reconsider the example in Fig. 3 from Section 2. Fig. 5 shows how to write the protocol transition with continuations by adding two new constructs to our notation. The *Suspend* statement (marked B) in the *ReadRequest* handler

---

3. Compare and Swap operation takes three arguments: an address, and "old" value and a "new" value. It performs the following action atomically: if the content at address is the same as old value, the content is updated with the new value and success is reported. Otherwise, failure is reported.
handles all messages that may arrive while waiting for a transition and need action. These messages require an analysis—unexpected messages may also arrive during the execution of these tasks. In a state machine, waiting for a succession of messages requires new intermediate states. By contrast, a primitive to the Stache protocol. The action codes effecting transitions between states are composed of messages. All other messages can be queued at the point captured by the continuation.

Eventually, a PutResponse message arrives for the block in transition, and the dispatch loop calls the corresponding handler in the subroutine state. The Resume action sequence (marked F) in the PutResponse handler restarts the continuation named in its argument. In this case, this continuation is the one captured at statement B, so statements C, D, and E are executed next in the current thread of control. On a Resume statement, the environment previously saved at a Suspend is restored, so the execution of the statements after a Suspend takes place in their original environment. Finally, the handler returns control back to the dispatch loop.

Using continuations in this manner, a programmer need not split the code in a handler into atomically executable fragments when waiting for a particular message. However, as discussed in Section 2, many protocol transitions are not simply a matter of awaiting a particular message—unexpected messages may also arrive during the transition and need action. These messages require an explicit intermediate state in which the programmer handles all messages that may arrive while waiting for a particular event.

An attractive benefit of continuations is that they allow reuse of such intermediate states between different transitions by providing a notion of calls between states. A subroutine state (such as AwaitPutResponse) is oblivious to the state from which it was entered and the state to which it goes next, since the continuation contains this information. Thus, the state can be a subroutine for all transitions that await and anticipate the same message(s) and perform the same actions in response to those messages. By contrast, a state machine requires each handler to have a distinct intermediate state to encode subsequent transitions. Subroutine calls permit significant code reuse in real protocols, as the action codes effecting transitions between states are frequently identical. For example, in the Stache protocol [13], the four different handlers that wait for a PutResponse message share a single subroutine state. More complex protocols (e.g., LCM in Section 4.1) presented even more code reuse opportunities. In effect, continuations turn a finite-state automaton into a pushdown automaton.

Another important benefit of continuations is that they can dynamically nest: a subroutine called from a Suspend can itself invoke another Suspend. A stack of continuations offers a convenient mechanism to record incomplete tasks. In a state machine, waiting for a succession of messages requires new intermediate states. By contrast, a function called from Suspend can itself suspend for another message. This provides a useful abstraction for processing a series of messages M1 and M2, since the code in AwaitM1 can execute a Suspend(WaitM2). For example, in the Stanford DASH protocol [14], a home node returns a Communicate that requires the writer to wait for Invalidation-Acks from the current readers. With this mechanism, the handler processing the response can directly Suspend to wait for the next acknowledgment.

Fig. 6 shows how continuations simplify adding a Compare&Swap primitive to the Stache protocol. The action code for this primitive invalidates outstanding copies by first invoking the transition ReadShared-To-Idle. Then, it performs the actual Compare&Swap operation. Similar changes are necessary in Idle and Exclusive. The code using continuations, unlike a pure state machine, forces the transitions within the IDE state by a subroutine-like mechanism, rather than encoding the pending Compare&Swap operation in the state until it can be executed in a transition into the Idle state.

A state, subroutine or normal, waits for a limited collection of messages. All other messages can be queued for delivery after a transition out of the state—or discarded or requeued, as a programmer chooses. Teapot does not impose a general solution to the problem of unintended messages, because different protocols have different needs. However, Teapot provides general mechanisms that permit a programmer to specify: which messages should be processed in a state, how to handle other messages that arrive, and what to do with these unintended messages. In a subroutine state (such as AwaitPutResponse in Fig. 5) that

4. The value of this facility become more obvious when writing handlers that naturally have many points of blocking communication. Some of these points may even occur inside nested control constructs.
that state. We exhibit the features of Teapot using an
on receipt of each message, should it arrive for a block in
specifies a set of message types and the actions to be taken
A Teapot program consists of a set of states; each state
waits for a reply message to complete a transition, a
programmer typically specifies a limited set of messages
that should be processed immediately and defers the others
(by enqueuing them) until the transition completes.
In order to convert a handler containing blocking calls
into nonblocking code, the Teapot compiler captures a
handler’s environment immediately before a **Suspend**,
using techniques similar to closures in functional program-
ing. The handler’s environment at the point of **Suspend**
is stored in a record and a pointer to it is stored within the
continuation variable. The original handler exits at this
point. The contents of this record are restored at a **Resume**
site, and the original handler is restarted immediately after the **Suspend** point.
Continuations in Teapot are different from those in
functional programming languages. In Teapot, calls to
**Suspend** can appear only in a handler’s body, and not in
a procedure called from the handler. This restriction
reduces the state that must be captured to only the local
variables of the handler, and lets the implementation avoid
the overhead of multiple stacks to support multiple
execution contexts. Another difference is that in Teapot,
the subroutine state named in a **Suspend** is not a function
that starts executing immediately, as is the case in
functional languages. Instead, the control implicitly passes
to a dispatch loop, which uses the named state to determine
which handler to apply to the next incoming message.
The Teapot compiler applies language-specific compiler
optimizations to reduce the performance overhead of using
continuations. The resultant code shows competitive per-
formance. We found that by using Teapot-generated
protocol code, applications generally ran with at most a
10 percent performance overhead compared to using hand-
written protocol code, even on a system with a fast
communication substrate (a CM-5 supercomputer), where
protocol software overhead has a higher impact on the
application performance. We have reported elsewhere (see
[15]) details on these compiler optimizations and the
performance data.

### 3.2 Language Features

A Teapot program consists of a set of states; each state
specifies a set of message types and the actions to be taken
on receipt of each message, should it arrive for a block in
that state. We exhibit the features of Teapot using an
example. The Teapot code in Fig. 7 implements coherence
actions in Stache protocol for a block in the **Exclusive** state at the
home node.

In Fig. 7, line 1 identifies the state for which the set of
handlers is specified. The action code for the message
**GET_RO_REQ** (lines 3-18) first sends a **PUT_DATA_REQ** message to the current owner (line 8). Note that the variable
**info** is a pointer to the directory data structure. It records
the identity of the requesting node (line 9). Next, it executes a **Suspend** statement. As discussed in the previous section,
it takes a program label (L), and an intermediate state (**Home_Excl_To_Sh**) which it visits “in transition.” The
second label, (L), specifies where execution should resume
upon return. When the control does resume at this point,
the handler sends the requested block to the set of
requestors (lines 14-17). Notice the use of an abstract type
to implement the iterator. Action code for the other
messages is specified similarly (but is omitted in the figure).
Finally, we specify the action for the **DEFAULT** messages
(lines 22-26). In this state, we have presumably listed the
action for all messages that can arrive; so arrival of any
other message is reported as a fatal error.

Fig. 8 shows the Teapot code executed when a
**PUT_DATA_RESP** message arrives in the intermediate state
**Home_Excl_To_Sh**. The handler receives the up-to-date
content of the cache block from the network, sets its own
state to **Home_RS**, and executes a **Resume** statement. The
**Resume** takes a continuation parameter (**C**) as an argument.
Note from line 1 in Fig. 8 that the continuation variable **C**
is a state parameter and is a part of the environment visible to
all the message handlers in that state. After the **Resume**
statement, **GET_RO_RESP** messages are sent to the set of
requesters (see Fig. 7 again, lines 14-17).

Teapot provides a mechanism for handling unexpected
messages by queuing. It does not solve the problem of
deadlocks directly, but facilitates deadlock detection via
verification. In lines 10-13 of Fig. 8, all messages not directly
handled (**DEFAULT**) are queued for later execution—these
messages are appropriately dispatched once the system
moves out of an intermediate (**transient**) state. Teapot relies
on a small amount of system-specific dispatch code to
deliver incoming network messages and previously queued
messages, based on a state-specific dispatch code and

---

**Fig. 6. Adding Compare&Swap to the basic Stache protocol. The code in this figure shows the action for the ReadShared state. Similar changes are required to Idle and Exclusive states.**

```plaintext
State ReadShared
Message Compare-N-Swap(Node n, Address a, Value oldVal, Value newVal)
   Suspend(L, ReadShared-To-Idle(L));
   If (*a == oldVal) Then
      *a := newVal;
      Send(n, Compare-N-Swap-Success);
   Else
      Send(n, Compare-N-Swap-Failure);
   EndIf;
   Exit;
```

6. Users must specify which states are transient.
DEFAULT messages in Fig. 7 flag an error because these messages cannot occur in a correctly functioning system.

As shown in these examples, Teapot handler bodies contain, in addition to continuation statements, conventional variable declarations and imperative constructs such as assignments, procedure call, conditionals, and loops. Handler bodies are typically short.

Teapot supports basic integer and boolean types and includes a facility for declaring abstract types and prototypes of functions that manipulate values of those types. Datatypes must be abstract because the Teapot system derives C code and MurΦ code from the same protocol specification. MurΦ’s types are far more limited than C’s [8]. Consider maintaining a list of processors in a data structure that supports operations such as include-sharer, delete-sharer, etc. A C implementation could keep a bit vector in a word and implement operations with efficient logical operators. MurΦ, on the other hand, represents the same information as an array of BitType, where BitType is an enumerated type of two values. To support both targets,
a Teapot program does not specify the implementation of a data structure. Instead, these programs declare abstract types, e.g., SharerList. We saw an example of an abstract type for iterators, SHARER_LIST_ITOR, in Fig. 7.

Consequently, a Teapot specification is neither a complete C program, nor a MurΦ program. The programmer must instantiate abstract data types by defining concrete representations and functions. In addition, some system-level issues, such as obtaining a proper dispatch loop for both the C and MurΦ versions have to be dealt with separately. Appendix A presents the Teapot grammar.

3.3 Verification Support
Several techniques can verify the correctness of a protocol by ensuring that it does not violate a set of invariants. Model checking by state space exploration is emerging as a popular technique for verifying hardware and software components. The MurΦ system [8], built by Dill et al. at Stanford University uses this technique. A MurΦ program specifies an initial state, a set of rules (ruleset), and a set of invariants. Rules fire only if their preconditions are satisfied. When a rule fires, a user-specified action code executes and the system’s state changes. MurΦ uses a Pascal-like input language to express conditions and actions. It selects the firing rule nondeterministically from the enabled rules, which permits simulation of asynchronous events. MurΦ explores all possible interleavings of events in a breadth-first fashion (although it has options for different search strategies) and checks that the invariants hold in every state. Should an assertion fail, it produces a trace of events leading to the erroneous state.

Teapot leverages off this system by automatically translating Teapot protocols into input code for the MurΦ verification system (in addition to generating C code for execution). This approach has two significant advantages, relative to testing a protocol using MurΦ directly:

- It saves a programmer the tedious process of specifying the same protocol once again in MurΦ’s input language (even the simplest coherence protocols can require several hundred lines of MurΦ code).
- By deriving both versions from a common Teapot source, it also eliminates potential discrepancies between the specification and the actual executable code.

One problem with model checking is limiting the size of the state space that must be explored. We must simulate a system configuration small enough that an exhaustive exploration is feasible. We tested several protocols using up to three nodes in the cluster, and having up to two shared addresses (see [15] for details on timings). While complete exploration could take several hours for configurations even this small, fortunately for us, error conditions were found rather quickly (within minutes). In our experience, simulating even a small model of the system was far more effective in detecting protocol errors, than with conventional debugging techniques (see next section for case studies).

Three basic components are required for verification with MurΦ: a MurΦ description of the protocol under test, MurΦ code implementing all types and subroutines used by the protocol, and a dispatch ruleset, which describes legal sequences of protocol events. While only the first component is generated by Teapot, examples of the remaining pieces are included with Teapot and can often be reused without modification. Some user intervention is required if new types or routines are added, or if the dispatch ruleset needs to be modified.

3.4 Overall System
The Teapot system is organized as follows. The Teapot compiler itself, implemented in SML/NJ, translates protocols written in Teapot language into either C or MurΦ, depending on whether an actual execution or verification is desired. Accordingly, it consists of two back ends, a C back-end and a MurΦ back-end. See Fig. 9.

To perform verification, a protocol programmer goes through the following steps. She uses the Teapot compiler to generate MurΦ code. As indicated in the previous subsection, she also provides the required support routines, rulesets and invariants. These are given to the MurΦ compiler, which generates code for state-space exploration. To obtain code for actual execution, a protocol programmer
uses the Teapot compiler to generate C code. She also provides support routines implemented as C routines. These are handed to a C compiler, which generates protocol object code. This object code can be linked into the rest of the runtime system.

4 Evaluation

The Teapot work was originally undertaken to aid protocol programmers for the Blizzard distributed shared memory system [16]. Blizzard exports a cache-coherence protocol programming interface to an application writer, so she can supply a coherence protocol that best suits the requirements of her application. After a few initial protocols (all simple variants of conventional shared memory protocols) were successfully developed using Teapot, the Blizzard team at Wisconsin wrote several, more complicated coherence protocols for their system. In this section we report on the experiences of the xFS team at University of California at Berkeley, which adopted Teapot to write the coherence protocol for their distributed file system. The discussion on xFS is joint work with Dahlin, Wang, and Anderson [17].

Initial implementation efforts for LCM and xFS began without the aid of domain-specific tools. The complexity of implementing the systems quickly became overwhelming in each case. LCM and xFS are both based on conceptually simple finite state machines, but these “paper” state machines expanded dramatically when, during implementation, complex transitions had to be broken into atomically-executable units and message reordering had to be considered. The subsequent state machines were quite complicated, and modifications in response to bugs often introduced new bugs in the process. The results were systems fraught with deadlocks, livelocks, core dumps, and most annoyingly, wrong answers.

Both groups experienced the same set of benefits upon adopting Teapot.

1. State machine management. Teapot’s domain-specific protocol-specification language allowed more succinct descriptions of the state machines in each case. Fewer states were required, since continuations often allowed similar states to be coalesced into a single subroutine state. Handler code legibility improved, since all the code for a given transition was kept contiguous.

2. Modularity. The implementors of both systems had unintentionally allowed system-level details to mingle with coherence protocol actions, and discovered that the modularity enforced by Teapot, in which protocol actions are kept entirely separate from system-level details, made it easier to specify and modify the coherence aspects of the system.

3. Debugging. The most helpful aspect of Teapot for both teams was its verification subsystem. Bugs that only arose after complex sequences of events were often found in minutes by Murphi. Many of these bugs were too convoluted to be detected via traditional debugging techniques or manual inspection of the code.

The following sections describe LCM and xFS, and present the details of a representative bug from each system. These bug descriptions are intended to convey the complexity of the errors that arose while implementing coherence protocols in realistic systems. These sections also describe the difficulties encountered during the implementations due to shortcomings in Teapot. Not surprisingly, the original Teapot design did not anticipate all of the requirements of other protocols in the context of Blizzard, much less those arising in a distributed file system context.

4.1 LCM

The Loosely Coherent Memory (LCM) coherence protocol [9] provides sequentially consistent distributed-shared memory as default, and is similar in many respects to protocols like DASH [14] and Stache [13]. The key difference is that LCM allows global memory to become temporarily inconsistent under program control. During such phases, a given data item may intentionally have different values on different processors. This makes management of shared data more difficult. Memory is returned to a globally consistent state by merging distinct versions of each data item and ensuring that all processors see the new values. This requires coordination among all processors in the system, and mixes computation (merge functions) with traditional protocol actions.

LCM implements the semantics of the data-parallel programming language C** [18] faster than conservative, compiler-implemented approaches. C** semantics specify that parallel function invocations on aggregate data do not interact. LCM enforces these semantics by keeping shared-data modifications private until all parallel invocations complete, then returns the system to a consistent state. Processes can still collaborate to produce values via a rich set of reduction operations (including user-specified reductions), but the results of these reductions are not available until after all parallel function invocations finish.

4.1.1 LCM and Teapot

The initial (non-Teapot) LCM implementation was never completely debugged. Even with the cleaner subsequent design made possible by Teapot, we uncovered a total of 25 errors using automatic verification. Each error was fixed as soon as it was detected and understood, and the verification step was repeated. Many of these were subtle bugs that were unlikely to occur often in practice, but were all the more dangerous as a result. Fig. 10 illustrates an LCM bug that is representative of those found through verification. Both parts of this figure show messages being exchanged between a pair of processors, with time increasing from top to bottom. In each case, a preceding exchange of messages (not shown) has left the cache (nonhome) side with the exclusive copy of a given coherence block.

In Fig. 10a, the caching processor performs an LCM modification of the block, creating a version that is inconsistent with respect to other copies in the system. However, since the cache side held the exclusive copy at the time it performed the modification, it first sends a copy of the block home. This data can be used by the home to respond to requests for the block from other processors. The
block is returned home via a PUT_MOD message when the cache side is finished. The second LCM modification then faults and requests the block back from the home. Messages have been reordered in the network such that the first to appear at the home is the request for data. The home detects the reordering, since it knows the requestor already has a copy of the block. The correct action in this case is to await the SHARE_DATA message, then satisfy the request. The home leaves the block in the Home_LCM state to denote the fact that at least one processor has created its own version of the block.

Initially, we thought the arrival of the GET_RO_REQ in the Home_Excl state always implied the message reordering scenario in Fig. 10a, and both the hand-written version of LCM and the first Teapot version encoded this assumption. Unfortunately, in the more complicated case shown in Fig. 10b, this caused the protocol to respond incorrectly. The home should instead await the PUT_DATA_RESP message, transition to the Home_Idle state, and satisfy the request. Correcting the protocol is straightforward once the two scenarios have been identified, but it is unreasonable to expect an unaided programmer to have foreseen such a bug, due to the complexity of the cases involved. Verification much better handles the task of enumerating all chains of protocol events and ensuring that they are processed properly.

Using Teapot, the new version of LCM was written, verified, and running applications in two weeks’ time. Only one bug was uncovered during field testing of the new protocol, and it occurred in a simple support routine that was intentionally not simulated. Also, because of Teapot, we were able to implement easily three variants of LCM: one that eagerly sends updates to consumers at the end of an LCM phase, another that manages extra, distributed copies of some data as a performance optimization, and a version that incorporates both of these features.

4.1.2 Teapot Shortcomings

While Teapot made it significantly easier to get LCM written and working, it fell short of our needs in several respects. One significant obstacle is Teapot’s inability to

---

7. This scenario arises in applications where a given processor handles several of a set of parallel tasks consecutively.

8. The routine was deemed to be too simple to hide any bugs.
perform actions across a set of blocks. A message handler, for example, can only update the state of the block to which a message is directed. In LCM, action must periodically be taken across a collection of blocks. For example, during a merge phase, a processor returns all modified blocks to their homes, where they are combined with copies from other processors. An event handler was written to carry out this flushing operation for a single block, but the handler must somehow be invoked for each block returned. As an application runs, the LCM protocol constructs a list of modified blocks that require flushing at the next reconciliation. This list is traversed when the reconciliation phase begins, and the appropriate event handler invoked on each block. Additional C code was written to traverse the list and invoke handlers in the executable version of the protocol, but this code is outside the scope of the Teapot protocol specification and therefore cannot be verified. The workaround in Teapot was to structure the Mur prote ruleset so that, during reconciliation, it invoked the handlers for each block in the list. This restructuring significantly increased the complexity of the ruleset and therefore the chances that it could contain an error.

Even without operations on sets of blocks, the ruleset for LCM was already much more complicated than those for our previous protocols. Unlike Stache, where any arbitrary stream of interleaved loads and stores to shared memory must be handled, LCM only properly handles stylized sequences of loads and stores. There are distinct phases that all processors must agree to initiate, in which only certain access patterns are legal. Encoding this into a ruleset was a lengthy, complicated, and potentially error-prone process, and represented a significant fraction of the work required to implement LCM. It would be preferable to generate such rulesets automatically from a high-level description of a protocol’s memory model, but we currently are unaware of any techniques for doing so.

The last shortcoming was relatively minor. Teapot currently does not allow the testing of a pair of expressions for equality. There were several places in the protocol where pairs of states or node identifiers needed to be compared, and an external routine had to be written to perform these tests. Future releases of Teapot will consider extending the language to rectify these simple omissions.

4.2 xFS

xFS, a network file system described in several previous papers [3], [10], is designed to eliminate all centralized bottlenecks and efficiently use all resources in a network of workstations. One of the most important features of xFS is its separation of data storage from data management. This separation, while offering superior performance and scalability compared to traditional file systems, also requires a more sophisticated cache coherence protocol. In addition, other aspects of the cluster file system environment—such as multilevel storage and reliability constraints—further complicate the system compared to more traditional DSM coherence protocols.

The three main components of an xFS system are the clients, the managers, and the storage servers. Under the xFS architecture, any machine can be responsible for caching, managing, or storing any piece of data or metadata by instantiating one or more of these subsystems. Fig. 11 shows a sample xFS installation.

Each of the three subsystems implements a specific interface. A client accepts file system requests from users, sends data to storage servers on writes, forwards reads to managers on cache misses, and receives replies from storage servers or other clients. It also answers cooperative cache forwarding requests from the manager by sending data to other clients. The job of the metadata manager is tracking locations of file data blocks and forwarding requests from clients to the appropriate destinations. Its functionality is similar to the directory manager in traditional DSM systems. Finally, the storage servers collectively provide the illusion of a striped network disk.

xFS employs a directory-based invalidate cache coherence protocol. This protocol, while similar to those seen in traditional DSM systems, exhibits four important differences that prevent xFS from using previously developed protocols and that complicate the design of xFS.

1. xFS separates data management from data storage. Although this separation allows better locality and more flexible configuration, it splits atomic operations into different phases that are more prone to races and deadlocks.
2. xFS manages more storage levels than traditional DSM systems. For example, it must maintain the coherence of the kernel caches, write-ahead logs, and secondary storage.
3. xFS must maintain reliable data storage in the face of node failures, requiring protocol modifications that do not apply to DSM systems. For example, a client must write its dirty data to storage servers before it can forward it to another client.
4. The xFS client is heavily multithreaded and it includes potentially blocking calls into the operating system, introducing more chances for synchronization errors not seen in DSM systems.

After several unsuccessful attempts at completing the cache coherence protocol using traditional development methods, the system was rewritten with Teapot. Although xFS was eventually implemented successfully, a number of bugs were discovered along the way.

Fig. 12 shows an example of a bug in an early version of the xFS protocol that would have been difficult to isolate via field testing, but which Mur easily discovered. In this version of the protocol, we saw no need for the manager to maintain sequence numbers for its outgoing messages. If a receiver of a manager request was not ready to act upon it, it simply queued it for later processing. Mur found the following deadlock bug.

Initially, client B is the sole cacher of a clean block.

1. Client C sends a read request to the manager.
2. The manager forwards the request to client B. To indicate that Client B should send the data to Client C via cooperative caching, the manager also updates its state to indicate that both client B and C are caching the data.
3. Meanwhile, client A sends a write request to the manager.
4. The manager sends a revoke request to client B, which arrives at client B before the previous forwarding message, invalidating its data.
5. The manager sends a second revoke request to client C, which client C queues, because its requested data has not arrived.
6. Client B sends a write request to the manager, which the manager queues, because its previously sent revoke message has not been acknowledged.
7. The delayed forward message from step 2 finally arrives, which client B queues, because its request to the manager has not been satisfied.

Now we have finally reached a deadlock: client A is waiting for the manager to complete the revoke operations; the manager is waiting for client C to acknowledge the revoke request; client C is waiting for client B to supply the desired data; and client B is waiting for the manager to process its write request. One solution is to use sequence numbers to order the outgoing messages for a particular block from the manager, so the sequence of events seen by any client is consistent with the manager's view.

4.2.1 Teapot Shortcomings
Teapot was designed and is best suited for DSM environments in which the primitives available to protocol handler
writers are limited and simple. The xFS coherence engine, on the other hand, must interact with other components of the system such as the kernel and the active message subsystem via more powerful operations such as system calls and thread synchronization. This difference in terms of the power and expressiveness of handler primitives has revealed some shortcomings of Teapot that were not apparent in its original application domain.

The first shortcoming is the lack of support for multithreading. An xFS client is heavily multithreaded to support concurrent requests from the network, but the coherence engine generated by Teapot has a large amount of global state and is difficult to make thread-safe. Transforming the resulting Teapot coherence engine into a monitor was unsuccessful, as subtle thread deadlocks occurred when different xFS threads enter the coherence engine and other xFS modules in different orders.

The second shortcoming concerns blocking operations on local nodes, which occur frequently in xFS coherence handlers. For example, when an xFS client needs to invalidate a cached file data block, it makes a system call to invalidate the data cached in the kernel. This system call might block, waiting for some other event that requires the attention of the coherence engine. Although Teapot provides good support for blocking operations waiting for remote messages, using the same mechanism to handle local blocking operations is tedious. In the above example, one must split the synchronous system call into asynchronous phases; invent a new node to represent the kernel, invent new states for the kernel node, invent new messages the kernel must accept and generate, and write code to tie all these elements together. Better support for local blocking operations would have significantly eased the xFS protocol implementation.

The third shortcoming concerns users’ inability to add new arguments to Teapot handlers. We were faced with the unpleasant dilemma of either modifying Teapot itself or simulating additional arguments via global variables. The former suggests a limitation of the model; the latter workaround is bad software engineering and, in particular, it makes the multithreading problem worse. A more severe restriction is Teapot’s lack of support for operations that affect blocks other than the block on which the current message arrives. The problem arises, for example, when servicing the read fault of one block by an xFS client requires the eviction of a different block. This is similar to the problem encountered by LCM during its merging phase.

5 RELATED WORK

The Teapot work most closely resembles the PCS system by Uehara et al. at the University of Tokyo [21]. They described a framework for writing coherence protocols for distributed file system caching. Unlike Teapot, they use an interpreted language, thus compromising efficiency. Like Teapot, they write protocol handlers with blocking primitives and transform the program into a message-passing style. Our work differs in several aspects. Teapot’s continuation-based semantic model is more general than PCS’s, which is a message-driven interpretation of a protocol specification. PCS’s application domain is less sensitive to protocol code efficiency, so they do not explore optimizations. Finally, we exploit verification technology by automatically generating an input specification for the Mur® verification system.

Synchronous programming languages, such as ESTEREL [22] and the Statecharts formalism [23], are useful for describing reactive systems and real-time applications. The most important commonality between these programming languages and Teapot is that they all are ways of expressing complicated finite-state machines more intuitively than a flat automaton. They all support some mechanism for composing smaller, simpler state machines at the language level. A compiler then converts this composition into a flat automaton, which the programmer never has to deal with directly. ESTEREL supports decomposition of a larger state machine into smaller, concurrently running state machines that communicate synchronously. Statecharts support the notions of depth and orthogonality to build large state machines out of smaller ones. Teapot manages the cross-product interaction (and the resulting state-space bloat) of explicit protocol states and pending events by factoring the pending events into states implicit in the continuation stack. Teapot shares another feature with ESTEREL and Statecharts in its support for automatic verification.

Teapot also differs significantly from synchronous languages in many important respects. A key distinction between Teapot and ESTEREL is that ESTEREL always describes finite state machines. Conceptually, Teapot continuations can be nested dynamically to arbitrary levels, and therefore Teapot programs are not equivalent to finite state machines. Similarly, Teapot queues are not bounded a priori. A bounded nesting of continuations can easily be simulated in ESTEREL by an appropriate sequencing of signals between interacting state machines. In our experience, most protocols can be written in a way that statically limits the nesting of continuations, so the ESTEREL is expressive enough in this regard; Teapot’s execution model simply does not require the boundedness. Queuing, being inherently an asynchronous operation, is more complicated for ESTEREL. An ESTEREL program would have to simulate all possible states of a queue to some limited length, a cumbersome programming effort. On the other hand, ESTEREL handles nondeterminacy in the ordering of events in a cleaner manner than Teapot. For example, suppose a node is waiting on two messages, A and B, in either order. Teapot will have to explicitly code the possibility of A arriving before B, and of B arriving before A. This can be handled more conveniently using ESTEREL’s parallel composition.

Teapot differs from synchronous languages in several other respects. It does not have a notion of time, so it is not suitable for programming real-time applications. The notion of concurrency in synchronous languages is different from that in Teapot. In synchronous languages, logical concurrency of state machines is a vehicle for expressing

9. Note that the model that is constructed for verification purposes will have to bound such dynamic behavior, as the model checker checks only a finite state space. It is the actual execution that need not be bounded a priori.
interacting subcomponents; such concurrency is later compiled away to obtain a single-thread program. A Teapot program logically specifies only one state machine. The need for concurrency arises because several such programs are required to run on the same processing resource—they have to interleave their execution (essentially as coroutines).

Wing et al. [24] present an eloquent case for using model-checking technology with complex software systems, such as a distributed file system coherence protocols. We also use model checking technology, but our primary focus is on a language for writing coherence protocols, and on deriving executable code as well as the verification system input from a single source. They write the input to the model checker separately from their code, which introduces the possibility of errors.

The design and implementation of domain-specific languages has spurred considerable interest in the systems programming community. Recent work includes instruction-set description languages [25] [26], a specification language for automatically generating network packet filters [27], and compiler optimizations for interface description languages [28].

6 Conclusion: Implications for Domain-Specific Languages for Systems Software

It would be gratuitous to reiterate the successes and shortcomings of Teapot. Instead, we present some generalized insight gained from building and using Teapot. Although our experience is with one domain-specific language, we hope that our observations will be useful for designers of other domain-specific languages, particularly for systems software.

6.1 How Big to Make the Language?

An important consideration when designing a domain-specific language is: how general should the language be? Teapot leans heavily to a minimal language and relies on externally written routines. For example, it has to call a function SameNode to compare two values of the type NODE, because we could not decide how far, if at all, we wanted to support equality on opaque types in the language. Another example is whether procedure calls should be a part of the language. If so, are there any restrictions to be observed in the code for the procedures? For example, Teapot does not allow a handler to suspend inside called procedures.

More comprehensive languages have the advantage that less code needs to be written in external routines. However, a larger language is harder to learn, harder to implement fully, and could be harder to optimize. While smallness has virtues, a designer should not go overboard and apply senseless restrictions. In Teapot, for example, most users were unhappy about the fixed set of arguments that appeared as handler parameters.

Capturing the commonly occurring programming scenarios is an important role of domain-specific languages. Teapot, for example, incorporates carefully designed abstractions for waiting for asynchronous messages. However, these abstractions were less effective at capturing the scenario of waiting for asynchronous events in general. This kind of waiting in xFS had to be cast into the waiting-for-messages idiom using extra messages. In hindsight, the language could have been designed to support asynchronous events, with messages as a special case of events.

For problem domains where it makes sense, it is imperative to think about automatic verification from the very beginning. In Teapot, for example, we maintained a clear distinction between opaque types and their implementation. In fact, the language has no mechanism to describe the implementation of opaque types. This was done so the verification system and C code could provide an implementation suitable for their purpose, rather than providing a common base implementation which may be poor for both purposes. An example of such an abstract type is a list of sharers, which is implemented using low-level bit manipulation in C, but using an array of enumerated type 0..1 in Murφ. The cost of this approach is that a programmer (not compiler writer) must supply the implementations, which, fortunately, are reusable.

6.2 Compiler Issues

Ideally, language users should only need to know the language definition, not the details of the language implementation. Even popular general-purpose languages fall short of this ideal by great distances, at least for systems software. We have three observations in this regard. First, a language’s storage allocation policy should be made clear—programmers generally like to know where in memory particular variables live and what their lifetime is. In Teapot, the storage for state parameters was not clearly defined. It was also not clear to programmers how the memory management of continuation records happened. In fact, in the current implementation, unless Suspend and Resume calls dynamically match, continuation records leak, as we do not provide garbage collection. Fortunately, most protocols naturally have such balanced Suspend and Resume paths.

Second, compiler optimizations should be explicitly specified and should be under user control. Even with all the virtues of verification, a systems programmer may need low-level debuggers (perhaps for reasons unrelated to the coherence protocol). A restructuring compiler such as Teapot’s makes the generated code harder to trace at runtime. Finally, despite these complications, we believe that aggressive optimizations are essential. In our experience, users are unwilling to compromise efficiency for ease of programming, particularly considering that speed is often the main purpose for distributing a computation.

6.3 Threads

As thread programming becomes commonplace, domain-specific language designers must pay close attention to thread support. Even when the language does not currently support threads, if it is successful, sooner or later users will want multithreading support. The DSL designer, due to her unique knowledge of the internals, should be prepared to provide recommendations, if not a full implementation, of thread support.

The first observation from our experience is that thread support cannot be treated as an afterthought; instead it must be an integral part of the early language design. When
we attempted to make Teapot thread-safe as an add-on, we quickly discovered that global state made this an error-prone process. Even though we only introduced a small number of coarse grain locks, they frequently led to subtle synchronization problems because these locks were not exposed at the interface level. They broke abstractions and could easily lead to deadlocks. The second observation concerns the different alternatives that can enable a module written in a domain-specific language to interact with other multithreaded components. We have found that a viable alternative to making Teapot thread-safe is to turn the generated code into a single threaded event loop [29]. Instead of allowing multiple threads to execute concurrently in the cache coherence state machine, these threads interact with the single thread of the state machine via events. This approach eliminates unnecessary thread synchronization inside the state machine.

6.4 Distribution and Cost of Entry

Most users are reluctant to even install a new programming language, much less learn it. Thus, designers of domain-specific languages should be prepared for considerable handholding: provide a very complete set of examples, documentation, and a distribution that builds out-of-the-box. The xFS group found that a set of complete examples was a crucial aid to adopting Teapot. However, Teapot faced two stumbling blocks: we asked our users to go pick up SML/NJ compiler from Bell Laboratories, and the MurΦ system from Stanford. Many people quit at this point, even when we offered to lead them through obstacles. Perhaps clever perl scripts could pick up the right software from web. Adding to our difficulties, all pieces of our system—SML compiler, MurΦ compiler, and the Teapot source—were constantly in flux, and it was very difficult to maintain version consistency. We see no easy way out of this situation. From the point of view of distribution, it would be best to provide everything in portable C code. However, without drawing upon previously distributed software, we could not have built Teapot in a reasonable amount of time.

6.5 A Spade is Not a General-Purpose Earth-Shattering Device

A tool-builder should be up front about what a tool does and does not do. Despite our efforts, several people thought of Teapot as a verification system, which it is not. In fact, we got an inquiry about Teapot that implied that we have discovered a more practical way of doing model checking than brute-force state-space exploration! Also, we note that Teapot is not directly suitable for describing hardware cache-coherence controllers because it permits unbounded levels of continuations. We were also asked why Teapot would not be suitable for model checking systems unrelated to cache-coherence. These observations become apparent when people forced us to think beyond the context of Blizzard style distributed shared memory systems. One should think carefully about a language’s or system’s restrictions and why they exist from the beginning, so as not to unnecessarily frustrate potential users.

Finally, we hope our work provides further and concrete evidence that it is better to build application-specific tools than to program complex systems with ad-hoc code. In our experience, it is more profitable to start with a focused domain-specific language or tool that solves a very specific problem to the satisfaction of a small user-community. Language extension and attempts at generalizing the application-domain should be considered only afterwards. Languages and tools with a large scope to begin with run the risk of being useful to no one, because they take much longer to design and implement, and ultimately be less useful to users than a more focused tool.

APPENDIX A

Teapot Grammar

```
program:
    modules protocol states
modules:
    [ module ]*
module:
    module id begin mod-decls end ;
mod-decls:
    [ mod-decl ]+
mod-decl:
    type id ;
    sub-decl
    const id : id ;
    sub-decl:
    function id ( sub-args_opt ) : id ;
    procedure id ( sub-args_opt ) ;
protocol:
    protocol id begin prot-decls_opt end ;
prot-decls:
    [ prot-decl ]+
prot-decl:
    var id : id ;
    const id := id ;
    state id ( state-args_opt ) transient_opt ;
    message id ;
states:
    [ state ]+
state:
    state id . id ( state-args_opt ) begin msgs end ;
state-args:
    state-arg [ ; state-arg ]+
state-arg:
    vars : id
msgs:
    [ msg ]+
msg:
    message id ( sub-args_opt ) block-decls_opt begin stmts end;
sub-args:
    sub-arg [ ; sub-arg ]+
sub-arg:
    var vars : id
    vars : id
block-decls:
    var [ var-decl ]+
```

ACKNOWLEDGMENTS

This work was performed while all the authors were affiliated with the University of Wisconsin at Madison. Much of the experience section was a result of efforts by the xFS team at University of California at Berkeley. The team members included Michael Dahlin, Randolph Wang, and Thomas Anderson. Mark Hill brought together the xFS and the Teapot teams. We thank Lalita Jagadeesan and one of the anonymous reviewers for helping us compare Teapot to the ESTEREL programming language.

This work was supported, in part, by Wright Laboratory Avionics Directorate, Air Force Material Command, United States Air Force, under Grant No. F33615-94-1-1525; by the Advanced Research Projects Agency (ARPA) Order No. B550; by the National Science Foundation (NSF) National Young Investigator Award No. CCR-9357779; and by NSF Grant No. MIP-9625558. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the United States Government.

Teapot is freely distributed. Please see the Teapot page for the latest version: http://www.cs.wisc.edu/~chandra/teapot/index.html, or contact one of the authors.

REFERENCES

Satish Chandra received his BTech degree in computer science from the Indian Institute of Technology, Kanpur, in 1991, and his MS and PhD degrees in computer science from the University of Wisconsin at Madison in 1992 and 1997, respectively. Since 1997, he has been a member of the technical staff at Bell laboratories, Lucent Technologies. His research interests include programming languages, compilers, computer architecture, and parallel and distributed computing. He is a member of the IEEE and the IEEE Computer Society.

Bradley Richards received the BA degree in computer science and physics from Gustavus Adolphus College, St. Peter, Minnesota, in 1988, and his MS and PhD degrees in computer science from the University of Wisconsin at Madison in 1992 and 1996, respectively. He is an assistant professor of computer science at Vassar College, Poughkeepsie, New York. His current research interests include parallel and distributed computing, models of concurrency, formal verification techniques, and programming languages.

James R. Larus received his AB degree from Harvard College and the MS and PhD degrees from the University of California at Berkeley. He was a professor of computer science at the University of Wisconsin at Madison from 1989–1999. His interests included programming languages, compilers, parallel programming, computer architecture, and program measurement. In 1998, he joined Microsoft Research as a senior researcher. He is a member of the IEEE and the IEEE Computer Society.