Teapot: Language Support for Writing Memory Coherence Protocols

Satish Chandra
Brad Richards
James R. Larus

Technical Report #1291

February 1996

Appears in *SIGPLAN Conference on Programming Language Design and Implementation (PLDI), May 1996*

Teapot: Language Support for Writing Memory Coherence Protocols

Satish Chandra, Brad Richards, and James R.Larus
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton St.
Madison, WI 53706 USA
{chandra, richards, larus}@cs.wisc.edu

Abstract

Recent shared-memory parallel computer systems offer the exciting possibility of customizing memory coherence protocols to fit an application's semantics and sharing patterns. Custom protocols have been used to achieve message-passing performance—while retaining the convenient programming model of a global address space—and to implement high-level language constructs. Unfortunately, coherence protocols written in a conventional language such as C are difficult to write, debug, understand, or modify. This paper describes Teapot, *a small, domain-specific language for writing coherence protocols. Teapot uses continuations to help reduce the complexity of writing protocols. Simple static analysis in the Teapot compiler eliminates much of the overhead of continuations and results in protocols that run nearly as fast as hand-written C code. A Teapot specification can be compiled both to an executable coherence protocol and to input for a model checking system, which permits the specification to be verified. We report our experiences coding and verifying several protocols written in Teapot, along with measurements of the overhead incurred by writing a protocol in a higher-level language.*

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, an NSF NYI Award CCR-9357779, NSF Grant MIP-9225097, DOE Grant DE-FG02-93ER25176, and donations from Digital Equipment Corporation and Sun Microsystems. The U.S. 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 U.S. Government.

1 Introduction

A shared-memory coherence protocol manages the replication of data, to ensure that a parallel program sees a consistent view of memory. In general, a coherence protocol ensures consistency when a processor writes to a shared location either by invalidating outstanding copies or by updating copies with the new value. A protocol can determine, to a large extent, the performance of a shared-memory program, since communication occurs through loads and stores to shared data. A mismatch between a protocol and an application's sharing pattern leads to excessive communication. For example, invalidation protocols perform poorly for producer-consumer sharing, since invalidating outstanding copies forces the consumers to re-request data, which requires up to four protocol messages for a small data transfer [7].

Recent shared-memory architectures provide, to varying degrees, the flexibility of tailoring a coherence protocol to better fit an application's needs. An example of such an architecture, the one on which this work is based, is the *Tempest* interface, a specification of the mechanisms necessary to implement shared-memory coherence protocols at user-level, i.e., as unpriviledged code running in an application program [14, 23]. With Tempest, an application programmer can either use a ready-made shared-memory protocol from a library or construct a customized shared-memory protocol that suits an application's needs, thereby achieving higher performance [11]. Compiler writers can exploit custom coherence policies as well. For example, a recent paper showed how a custom protocol (LCM) enables a compiler to use a fine-grain, copy-on-write mechanism to implement a language's copy-in, copy-out semantics, without being limited by conservative pointer analysis [18].

Unfortunately, Tempest mechanisms are low level and protocols can be difficult to write, debug, and modify. Writing coherence protocols in an ad-hoc manner (as C programs) artificially limits the number of programmers willing or able to develop custom protocols. This paper

addresses the problem with a new language, called Teapot, that supports both writing and verifying coherence protocols. By reducing the complexity of developing new protocols, Teapot increases the attractiveness of systems, such as Stanford Flash [17], Wisconsin Typhoon [23], and Blizzard [24], that implement protocols in software. Protocols written in Teapot could also be used for distributed object systems, such as Multipol [6], CRL [16], and Orca [3].

The main difficulty in writing a coherence protocol is the requirement that all protocol actions (code executed in response to a protocol event, such as a read miss or an incoming message) run to completion before relinquishing the processor to execute protocol code for other cache blocks. Action code, however, frequently needs to engage in blocking communication, e.g., sending a request message to another node and awaiting a reply. To circumvent this problem, protocol programmers introduce intermediate states in their finite state machines. A protocol awaits a reply message by shifting into an intermediate state and relinquishing the processor. The intermediate state handles the message on its arrival, and then goes to the destination state (one that reflects the overall effect of the protocol event). The resulting proliferation of states complicates protocols, because non-atomic transitions force a programmer to reason about the interaction between all messages and each state. In practice, protocols written in this style are difficult to debug or modify. Detailed discussion of coherence protocols and their complications appears in Section 2.

Teapot allows a programmer to suspend an action handler and await an asynchronous event by calling a waiting function with the current continuation (Section 3 and Section 4). Teapot also permits the programmer to concisely specify how to handle other messages that arrive while waiting. The Teapot compiler transforms the handler with suspending calls into atomically executable pieces that can run without causing deadlock. Analysis and optimization in the Teapot compiler (Section 5) eliminates most of the overhead incurred by writing protocols at a higher level of abstraction.

Section 6 presents our experience with using Teapot to write several protocols for the Blizzard system [24]. With optimization disabled, our compiler generates protocol code in C that runs application programs within 13% of the speed of a hand-written protocol. With optimization enabled, the performance is (except for one program) within 10% of the corresponding hand-written protocol, which is very attractive given the vastly superior protocol writing environment.

Even with better programming abstractions, protocols can be difficult to write correctly. Field testing cannot ensure correctness since protocols contain complex timing-dependent paths. Although completely verifying a nontrivial protocol is difficult, model checking through state space exploration has emerged as a viable debugging aid for complex protocols [8, 9, 22, 28]. In addition to executable C code, the back-end of the Teapot compiler can generate input for the MurΦ [9] verification system (Section 7). Protocol verification has been one of the greatest benefits of this system, as it has reduced the time to develop a complex protocol by an order of magnitude.

2 Why are coherence protocols hard?

A shared-memory system can be built on two basic mechanisms. The first mechanism, *access control*, allows the system to control access to memory by permitting read and write accesses only for valid, cached data. Reading or writing an invalid location or writing a valid but read-only location must cause an *access fault* and invoke the coherence protocol. The second mechanism, *communication*, enables a system to transfer control information and data among processors.

A protocol comes into play at an access fault. It must satisfy the memory access by bringing the data to the faulting processor's memory. In many protocols, each block of shared data has a *home node* that coordinates accesses to the block. The accessing processor sends a request to the referenced block's home processor, which performs bookkeeping and returns the data. Once a processor obtains the data, it caches a copy, which can be subsequently accessed until it is *invalidated*. Many protocols, for example our Stache protocol [23], enforce coherence by permitting only a single writer (or multiple readers) to a block. When a home node receives a request for a writable copy of a block, it invalidates the outstanding copies before returning the block. The memory reference and the request and invalidation messages are *protocol events*, which cause transitions in a protocol state machine.

Both the home node and caching processors record state for a block. At a protocol event for block B, the protocol consults B's state to determine an action, which may send messages and update both the state and the contents of the block in memory. For example, Figure 1 depicts states in a simple protocol for a block at a non-home node (Figure 2 shows the corresponding home node state machine.) Consider a block that is initially *Invalid*. When the processor

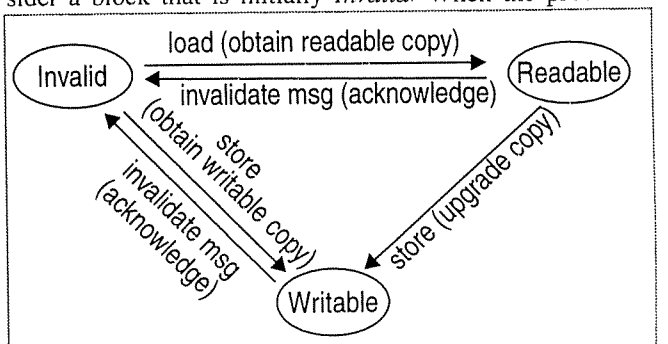


Figure 1: Idealized protocol state machine (non-home side). Transition are labeled with cause and, in parentheses, action.

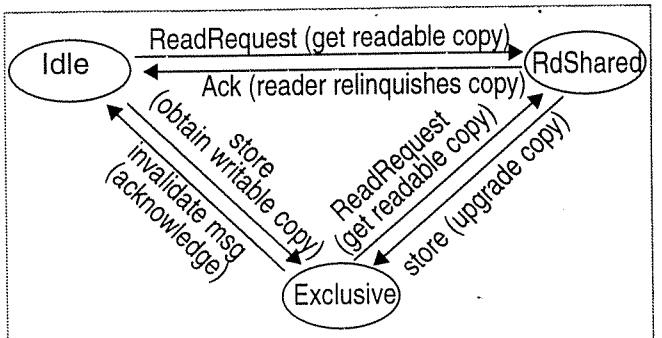


Figure 2: Idealized protocol state machine (home side). Transitions are labeled with cause and, in parenthesis, action.

reads the block, the protocol obtains a read-only copy from the home node and changes its state to *Readable*. Later, the home node may invalidate the copy with an *Invalidate* message, which causes a transition to state *Invalid*. Simultaneously, the block's access permissions are changed so subsequent reads or writes cause an access fault.

In general, actions execute on transitions between states, which are caused by protocol events. Actions may send messages to other processors, await their replies, update protocol-specific information, and change access permissions. The exact states, transitions, and actions depend on the coherence algorithm and the memory consistency model.

To illustrate the complications in writing a protocol, consider the home-side state machine of the Stache protocol (Figure 2). Conceptually, the protocol requires only three states: *Idle*, *ReadShared*, and *Exclusive* (a non-home processor currently has the single valid copy.) Although transitions appear as *atomic*, state changes in response to protocol events cannot be atomic. Consider the transition from *Exclusive* to *ReadShared*, which responds to a read request by a processor. This transition can complete only when the block's previous owner relinquishes it. Conceptually, the action for this transition sends an invalidation message and awaits the block's new value (Figure 3a). However, to avoid deadlock in a real system, protocol handlers must run to completion and terminate. Handlers cannot wait on an asynchronous event, such as a message arrival. Only the automaton can wait for a protocol event to cause a transition. Hence, after sending the invalidate message, the *ReadRequest* handler changes to an intermediate state (*Excl-To-ReadShared*) and terminates. When the *PutResponse* subsequently arrives, the transition completes by changing to state *ReadShared* (Figure 3b). Other states also require intermediate states for their transitions. Figure 4 shows the new, more complex state machine—which is still a simplification of the actual protocol.

Intermediate states complicate programming because they make transitions non-atomic. While in an intermediate state, the protocol may receive messages other than the expected reply message. For example, the state *Excl-To-ReadShared* waits for a *PutResponse* message. Before that message arrives, the home node may also attempt to write the same block and send a *WriteFault* message. The protocol programmer must consider these possibilities and provide the *Excl-To-ReadShared* state with a suitable action.

One unattractive approach is to receive these messages and encode a block's history, including pending actions, in its state. For example, a block in state *Excl-To-ReadShared* receives a *WriteFault* message and moves to another state *Excl-To-ReadShared-Pending-WriteFault*. When the *ReadShared* transition completes, the protocol processes the pending *WriteFault*. Because this approach greatly expands the state space, protocols either encode this information in an auxiliary data structure, negatively acknowledge (nack) unexpected messages, or queue unexpected messages for later processing. All three approaches have disadvantages. An auxiliary data structure complicates programming, because a programmer must remain aware that a block's state is split between two representations. Nacks can lead to deadlock, so they must be employed carefully. And, queuing requires additional memory. Teapot offers all three options, but advocates queuing unexpected messages.

Message reordering in a network further adds to the complexity, because messages may arrive in an unexpected order. For example, a *ReadRequest* from a processor that already has a readable copy cannot be ignored or treated as an error. The processor may have returned its copy with a *PutNoData* message and subsequently requested a readable copy with a *ReadRequest*. If messages can pass each other,

(a)

```
State Exclusive
   Message ReadRequest(Node R)
       Send(Owner, PutRequest);
       AWAIT PutResponse;
       SendSharedCopy(R);
       State := ReadShared;
       exit;
```

(b)

```
State Exclusive
   Message ReadRequest(Node R)
       Send(Owner, PutRequest);
       sharer := R;
       State := Excl-To-ReadShared;
       exit;

State Excl-To-ReadShared
   Message PutResponse(Node R)
       SendSharedCopy(sharer);
       State := ReadShared;
       exit;
```

Figure 3: The action in box (a) executes the exclusive to shared transition using synchronous (blocking) communication, which could deadlock on a real system. The action in box (b) shows an approach to avoiding synchronous communication: it introduces an intermediate state, which waits for a reply message.

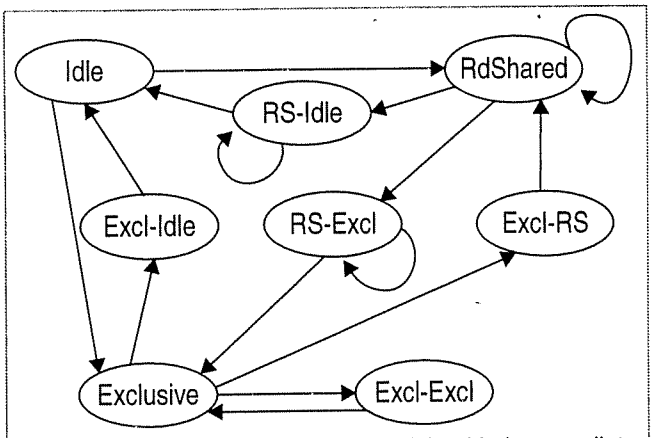


Figure 4: State machine (home side) with intermediate states necessary to avoid synchronous communication.

the seemingly gratuitious *ReadRequest* must be retained and processed after the *PutNoData* message. Teapot, by default, queues such messages for processing at a later time.

Another important consideration is that a protocol can be difficult to modify, although this is a desirable way of developing a custom protocol. Consider adding a *Compare&Swap* primitive to the basic protocol. This primitive is a minor variation of a *WriteRequest* that also executes the compare and swap operation at a block's home node once the block becomes *Idle*. Tracking a pending *Compare&Swap* complicates nearly every transition in a home node state machine. The state machine-based implementation needs to test for this condition at 14 different places.[1]

3 Continuations and Protocols

This section describes our approach to writing handlers. In general, any mechanism that permits multiple execution contexts to coexist can solve the problem of synchronous communication inside a handler without introducing explicit intermediate states. Continuations, coroutines, and threads[2] are three such mechanisms.

Using threads, a protocol event handler can be launched in a new thread, which yields control at a synchronous communication point. When the subsequent response message arrives, the thread is re-scheduled and runs to completion (or to a later synchronous communication point). Coroutines can be used in a similar way.

Continuations can also support synchronous communication in the following way. Recall that a continuation captures the execution state of the current computation, and passes the encapsulated state as an object to another computation. The second computation can then resume the captured computation, often passing along a value.

Consider the example from the previous section. Figure 5 shows how to write the protocol transition with continuations. The Teapot **Suspend** statement in the *ReadRequest* handler is similar to a **callcc**. It passes the current continuation (**L**, the first argument to **Suspend**) to the function that is its second argument. The Teapot **Resume** statement is similar to a **throw**, except that it does not return a value.[3] The **Suspend** statement captures its environment (the program position, as well as local variables) at the call point in its first argument, switches to the *subroutine state* specified by the second argument, and passes the continuation. At this point, the handler yields control (to a dispatch loop), until an action in response to a message to the subroutine state causes the suspended handler to resume. Subroutine states are parameterized by a continuation (e.g., the **Cont L** parameter in Figure 5b), which is part of the environment of all handlers in the state. A subroutine state differs from a normal state in two ways: it is entered synchronously by a **Suspend** command in another handler, and it can execute a **Resume** command to restore the environment and transfer control to the program point captured by the continuation.

Using these mechanisms, 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 the notion of calls between states. A sub-

(a)
```
State Exclusive
    Message ReadRequest (Node R)
        Send(Owner,PutRequest);
        Suspend(L,AwaitPutResponse{L});
        SendSharedCopy(R);
        State := ReadShared;
        exit;
    ...
```

(b)
```
State AwaitPutResponse(Cont L)
    Message PutResponse
        Resume(L);
        exit;
    ...
```

Figure 5: Using continuations to avoid introducing new states. State **AwaitPutResponse** is used as a subroutine state.

1. In C code, we used a flag in the protocol state associated with a block, which is basically a doubling of existing states. Maintaining flags, however, is more tractable from a programmer's point of view.

2. These are light-weight, self-scheduling threads that yield control to each other (through a scheduler), rather than true concurrent threads that can potentially execute on multiple CPUs.

3. This is in keeping with the imperative nature of Teapot. Each block has a global "info" area available, which can be used to communicate values.

routine state (such as **AwaitPutResponse**) is oblivious to the state from which it was invoked 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, the four different handlers that wait for a *PutResponse* message share a single subroutine state. More complex protocols (LCM in Section 6) presented even more code reuse opportunities. In effect, continuations turn a finite-state automaton into a push-down automaton.

Continuations can nest: a subroutine called from a **Suspend** can itself invoke another **Suspend**. A stack[1] 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(AwaitM2)**. For example, in the Stanford DASH coherence protocol [19], a home node returns a *WriteResponse* 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. This feature was used several times in the LCM protocol.

Figure 6 shows how continuations simplify adding a *Compare&Swap* primitive to the Stache protocol. It contains code that handles a message to a home node for a block in *ReadShared* state. The handler first invalidates outstanding copies and then invokes the transition *ReadShared-To-Idle*. Then, it performs the actual *Compare&Swap* operation. Similar changes (not shown) are necessary in states *Idle* and *Exclusive*. If the *Compare&Swap* message arrives in any other state, Teapot automatically queues it until the block enters a state that can process the message. The Teapot code, unlike a state machine, forces the transition to the *Idle* 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.

```
State ReadShared
   Message Compare-N-Swap(Node n,
                          Address a,
                          Value old_val,
                          Value new_val)
      ...
      Suspend(L, ReadShared-To-Idle{L});
      If (*a == old_val) Then
          *a := new_val;
          Send(n, Compare-N-Swap-Success);
      Else
          Send(n, Compare-N-Swap-Failure);
      Endif
      exit;
```

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

A state, subroutine or normal, waits for and processes a limited collection of messages. All other messages can be queued for delivery after a transition out of the state—or discarded or nack'ed, 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 Figure 5) that 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 non-blocking code, the Teapot compiler captures a handler's environment immediately before a **Suspend** (Section 5). Teapot continuations are not fully general: calls to **Suspend** can appear only in a handler's body and not in an external procedure called from a handler. This reduces the state that must be captured to only the local variables of the calling routine, and thus facilitates optimizations.

In principle, we could have achieved the same effect by providing a light-weight threads package to the programmer. However, continuations are a higher-level language feature and lend themselves more readily to compiler optimizations.

4 Teapot Language

Teapot is best introduced by a code fragment from the Stache protocol (Figure 7 and Figure 8). Note that this is the *complete, executable* code for those states: no details are elided. This code implements the protocol messages governing a block in *ReadOnly* state on a non-home processor. If the processor attempts to write to the block, Tempest sends a protocol event **WR_RO_FAULT**. The action for this event first sends an **UPGRADE_REQ** to the home node and then **Suspend**s, calling the subroutine state **Cache_RO_To_RW**. Later the code for

1. This is not to imply that **Resumes** should dynamically match the corresponding **Suspends**; however, all **Suspends** must eventually be **Resumed** (or otherwise deallocated) to prevent memory leaks, as there is no built-in garbage collection.

```
State Stache.Cache_ReadOnly{}
Begin
    Message WR_RO_FAULT (id: ID;
                         Var info: INFO;
                         home: NODE)
    Begin
        Send(home, UPGRADE_REQ, id);
        Suspend(L, Cache_RO_To_RW{L});
        WakeUp(id);
    End;

    Message PUT_NO_DATA_REQ (id:ID;
                             Var info:INFO;
                             home: NODE)
    Begin
        Send(home, PUT_NO_DATA_RESP, id);
        SetState(info, Cache_Inv{});
        AccessChange(id, Blk_Invalidate);
    End;

    Message DEFAULT(id: ID;
                    Var info: INFO;
                    home: NODE)
    Begin
        Error("Invalid msg %s to Cache_RO",
              Msg_To_Str(MessageTag));
    End;
End;
```

Figure 7: Teapot example from the Stache protocol. Continued in Figure 8.

```
State Stache.Cache_RO_To_RW{C:CONT}
Begin
    Message UPGRADE_ACK (id: ID;
                         Var info: INFO;
                         home: NODE)
    Begin
        SetState(info, Cache_RW{});
        AccessChange(id, Blk_Upgrade_RW);
        Resume(C);
    End;

    Message GET_RW_RESP (id: ID;
                         Var info: INFO;
                         home: NODE)
    Begin
        RecvData(id, Blk_Upgrade_RW);
        SetState(info, Cache_RW{});
        Resume(C);
    End;

    Message DEFAULT(id: ID;
                    Var info: INFO;
                    home: NODE)
    Begin
        Enqueue(MessageTag, id, info, home);
    End;
End;
```

Figure 8: Teapot example continued from Figure 7.

`Cache_RO_To_RW` executes a **Resume** statement, which restarts the suspended handler, which in turn restarts (wakes up) the original computation thread that faulted.

State `Cache_ReadOnly` (Figure 7) explicitly handles only the `WR_RO_FAULT` and `PUT_NO_DATA_REQ` messages. All other messages pass to the **DEFAULT** code, which raises an error since other messages are erroneous in this state. Figure 8 shows the subroutine state `Cache_RO_To_RW`. The home node responds to an `UPGRADE_REQ` with one of two messages, hence the state must be prepared to handle both. In this state, the **DEFAULT** case enqueues all other message for later processing.

The operation **Resume(C)** takes a continuation and restarts the suspended handler (after restoring its environment). Notice the continuation parameter (`C`) in this state's definition. This argument is bound to the continuation passed at the **Suspend** in the previous handler (State `Cache_ReadOnly`, Message `WR_RO_FAULT`, in our case). If a message handler in this state contained a **Suspend** operation, the variable `C` would also be saved and restored. In this way, **Suspend**s nest dynamically, like `callcc` in ML or Scheme.

In addition to continuation statements, Teapot handler bodies can contain conventional imperative constructs: assignments, procedure calls, conditionals and while loops. Handler bodies are typically short.

Teapot supports basic integer and boolean types and includes a facility for declaring compound 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 [9]. 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 operations. MurΦ, on the other hand, represents the same information as an array of BitType, where BitType is a 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. As a result, 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 function for both the C and the MurΦ code, are omitted from this paper since they are routine and reusable for different protocols.

Appendix A presents the Teapot grammar.

5 Compilation

Teapot code could be compiled to use light-weight threads in which **Suspend**s and **Resume**s would cause saves and restores of thread contexts. Since we restricted our input language, the generality of threads is unnecessary. First, **Suspend**s occur only at statement level (by contrast call/cc can occur anywhere in an expression). Second, a **Suspend** cannot appear in a function called from the main handler. Given these restrictions, a handler written with **Suspend**s and **Resume**s can be directly compiled into non-blocking event handlers, without multiple stacks.

To illustrate the process, we will apply source-level transformation to the handler in Figure 9. The compiler first transforms the handler into two routines (Figure 10), the first of which includes code up to the **Suspend** and the second that includes the remaining code. A **Resume** statement simply extracts the function pointer from the continuation record and calls the second function. This transformation works even if **Suspend** statements occur within control structures (nested loops and conditionals). The first fragment runs through the **Suspend** then exits the handler routine. The code for the second part starts after the **Suspend.** Well-structured programs—all Teapot programs—can always be split this way.

An optimization is to save and restore in the continuation only values that are referenced after the **Suspend.** Often in a handler, no values are saved and restored, so that a continuation can be statically allocated and used by all handler invocations. Furthermore, the compiler detects if a constant continuation reaches a particular **Resume** site. If so, the code from the handler can be in-lined at the **Resume** site. This optimization is similar to β-contraction, as discussed by Appel [2], and has proved effective in our experiments .

6 Case Studies

Previous sections presented several examples from the Stache protocol. We wrote the Stache protocol in Teapot (600 lines, which compiles to 1000 lines of C code) and compared its performance against the original, hand-written, state machine implementation (approximately 1000 lines of C). Table 1 contains performance evaluation of four benchmarks running under the Stache protocol (on a 32 processor CM-5 running Blizzard-E [24]). The second column reports the application running time (millions of cycles) for the hand-written C protocol. The next two columns report times for unoptimized and optimized Teapot protocols. In the unoptimized results, the compiler performs live variable analysis but not the constant continuation optimization. The optimized numbers show the times when the compiler performs both live variable analysis and the constant continuation optimization. The next column reports the number of continuation and queue records allocated and freed on all nodes. The last column reports the average time across nodes spent waiting for faults and message handlers in the C state machine. This number is a measure of the communication overhead of a program. It, however, is not a true measure of the time spent in protocol processing, because running a handler while waiting for an earlier fault adds no overhead. This number puts the performance overhead of Teapot in proper perspective.

```
Message HANDLER1(arg1: T1; arg2: T2)
Var
    l1,l2: T3;
Begin
    stmt1
    stmt2
    Suspend(L, NewState{L});
    stmt3
    stmt4
End;
```

Figure 9: A sample handler, with a `Suspend` point. The compiled code is in Figure 10.

```
(a)
Message HANDLER1(arg1: T1; arg2: T2)
Var
    l1,l2: T3;
Begin
    stmt1
    stmt2
    L := AllocContinuation();
    L.funcPtr := HANDLER1_after_L;
    Save arg1, arg2, l1, l2 in L;
    Put L in environment;
    State := NewState;
    exit;
End;

(b)
Message HANDLER1_after_L(L: CONT)
Var
    arg1: T1;
    arg2: T2;
    l1,l2: T3;
Begin
    Restore arg1,arg2,l1,l2 from L;
    FreeContinuation(L);
    stmt3
    stmt4
    exit;
End;
```

Figure 10: Code generated for the handler in Figure 9.

The optimized Stache protocol ran 5-10% slower than the C version. The constant continuation optimization, however, effectively reduced the number of continuations and improved execution time of programs that allocate a disproportionate number of continuations on a few processors (because of memory reference patterns). We will shortly return to the issue of performance difference between the Teapot and state machine versions.

We believe that most new protocols will be variants of existing ones. For example, we implemented a variant of the Stache protocol that attempts to overlap the latency of acquiring a writable copy of a cache block with future computation by buffering writes until a synchronization point. The modifiation to Stache code involved adding 4 new states, 4 new message types, and some support routines. This protocol requires an application to have the synchronization needed by the weakly consistent memory model [1]. Since we did not have a state machine-based implementation, we are unable to present comparative performance data for this protocol.

LCM is a far more complex protocol [18]. It exploits controlled inconsistency in phases of parallel programs and has been used as run-time support for languages that require copy-in-copy-out semantics for parallel loops. When a program enters an LCM phase, each processor can obtain a copy of a location that is not kept consistent. A node can access its copy without affecting another processor. At the end of the LCM phase, each node with a copy reconciles its modifications with other nodes, so that the system returns to a consistent state. Figure 11 shows how Teapot facilitates handling a complex network reodering problem that arises in the LCM protocol.

The LCM protocol in Teapot (1500 lines) compiled to approximately 2300 lines of C code. The state machine implementation of LCM protocol required approximately 2500 lines of C. Table 2 contains performance numbers for three benchmarks running under the LCM protocol. With optimizations, the LCM protocol performed comparably with the state machine version in most cases tested.

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 (LCM-Update), another that manages multiple, distributed copies of some data as a performance optimization (LCM-Mcc), and a version of LCM that incorporates both (LCM-Both) of these changes. Again, equivalent state machine versions of these protocols were not available for a performance comparison.

We found it very difficult to isolate the factors that would account for the performance difference between the Teapot and state machine protocols. In particular, CM-5 processors have small (64Kb), unified, direct-mapped caches, which can exaggerate the effect of small increases in code and local data. In addition, the SPARC register windows can penalize Teapot handlers since they add a level of indirect function call at all handlers[1].

To understand better the performance differences, we used a detailed architectural simulator of a multiprocessor that implements the Tempest interface. The simulated machine differs from the CM-5: it has larger (256Kb) data caches and unlimited register windows. Experiments with the Stache protocol showed that Teapot versions were consistently within 5% of the execution times of the state machine versions. Simulator statistics also show that event counts and the times spent in the two versions are comparable. Teapot overheads in message handler invocations account for the remaining difference.

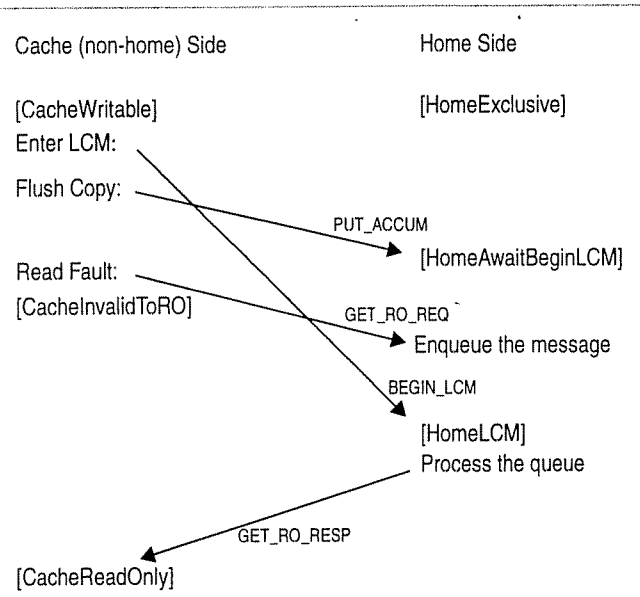
1. In the LCM benchmark that performed worst, the performance difference was reduced to within 6% by avoiding (via hand-coding) the use of register windows at one call site.

Table 1: Performance of Teapot system with Stache protocol.

Benchmark	Execution Time in cycles (% increase over C code)			Allocs in Opt/ Allocs in UnOpt	Fault time
	C State Machine	Teapot Unoptimized	Teapot Optimized		
gauss	1930 M	2150 M (11.4%)	2050 M (6.2%)	65.7K / 551K	40%
appbt	1860 M	2100 M (13%)	1990 M (7%)	19.9K / 1197K	36%
shallow	1160 M	1310 M (13%)	1280 M (10%)	0.3K / 1001 K	44%
mp3d	2210 M	2340 M (5.9%)	2320 M (5%)	443 K / 3249 K	72%

Table 2: Performance of Teapot system with LCM protocol.

Benchmark	Execution Time in cycles (% increase over C code)			Allocs in Opt/ Allocs in UnOpt	Fault time
	C State Machine	Teapot Unoptimized	Teapot Optimized		
adaptive	3301 M	3440 M (4.2%)	3376 M (2.3%)	124 K / 4410 K	28%
stencil	3717 M	4120 M (10.8%)	3859 M (3.8%)	3347 K / 7452 K	63%
unstruct	1431 M	1710 M (19.4%)	1666 M (16.4%)	62 K / 2572 K	38%




```
State LCM.Home_Excl{}
   Message PUT_ACCUM(...)
   Begin
      ...
      Send(src, PUT_ACCUM_ACK);
      Suspend(L,Home_Await_BEGIN_LCM{L});
      State := Home_LCM;
      ...
   End;
   ...
State LCM.Home_Await_BEGIN_LCM{C:CONT}
   Message BEGIN_LCM(...)
   Begin
      ...
      Resume(C);
   End;

   Message DEFAULT(...)
   Begin
      Enqueue(...)
   End;
```

Figure 11: This figure shows a network reordering problem occuring at several places in LCM. The cache (non-home) side sends the home a BEGIN_LCM message indicating that it is entering LCM phase. The message reaches the home after two other messages. The Teapot code on the right handles this scenario by going to an AwaitBeginLCM state. Note the queuing of GET_RO_REQ.

7 Verification

Several techniques can verify the correctness of a protocol by ensuring that it does not violate a set of invariants. Model checking by exhaustive state-space exploration is a popular technique in hardware cache-coherence community. The MurΦ system, built by Dill et al. at Stanford University, uses this technique. A MurΦ program specifies an initial state, a set of rules, and a set of invariants. Rules fire only if their preconditions are satisfied. When a rule fires, an 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 non-deterministically 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, MurΦ produces a trace of events leading to the erroneous state.

In general, MurΦ requires a programmer to write a protocol twice, once in an executable form and once in MurΦ's specification language. Writing a MurΦ specification requires significant effort. Our hand-coded specification of the Stache protocol was approximately 800 lines of MurΦ code. In addition, verifying a specification—rather than an executable protocol—can hide errors arising from the differences between the two.

To solve this problem, Teapot automatically generates a MurΦ specification from a Teapot protocol. Since a single source produces both verification and executable code, the MurΦ specification accurately captures the behavior of the executable code. In addition, Teapot saves the effort of writing a separate specification. However, a protocol writer must supply support routines that define data structures in MurΦ's input language and an event generation loop that generates a random sequence of events for which the protocol must work corectly. For example, in the Stache protocol, each node should process any stream of loads and stores to any shared addresses. For the Buffered-write protocol, each node must handle synchronization operations randomly interleaved with the loads and stores. To further check the correctness of values in the shared memory, a more stylized event generation loop is necessary, as the values will be consistent only if loads and stores obey a discipline [1] with respect to synchronization operations. Event generation for Stache and Buffered-write protocols required about 50 and 100 lines respectively of MurΦ code. LCM protocol event generation is quite complicated—it took about 400 lines of MurΦ code.

One problem with model checking is limiting the size of the state space that must be explored. In general, we simulated a minimal machine with 2 processor nodes and 2 shared memory addresses. Also, our verifications did not test actual data values. We currently verify that a protocol does not deadlock and that it does not receive a message that is not anticipated in a given state. Additional assertions can be verified as needed, but have not proven necessary.

Our experience with MurΦ has been very good. It found errors in a reasonable amount of CPU time (typically within an hour on a 66 Mhz SparcStation with 150M memory). It even uncovered an unsuspected protocol bug in a heavily-used implementation of the Stache protocol, which could

occur under a particular interleaving of messages in the network. Table 3 lists the verification times on a 66 MHz Sparc with 150M of memory for each of the protocols we wrote.

Table 3: Protocol verification times

Protocol	Configuration	Time Taken
Stache	2 nodes, 2 addresses 1 reordering max[a]	4900 seconds
Buffered-Write	2 nodes, 1 address 1 reordering max	302 seconds
LCM Simple	2 nodes, 1 address 1 reordering max	11515 seconds
LCM Mcc	2 nodes, 1 address 1 reodering max	5804 seconds
LCM Update	2 nodes, 1 address 1 reordering max	8745 seconds
LCM Both	2 nodes, 1 address 1 reodering max	1104 seconds

a. Out-of-order messages increase the number of states that MurΦ has to explore. We limited the amount of reodering in the simulated network, because unrestricted reordering (i.e., any number of later messages along a channel can cross an earlier message) led to impractical simulation sizes.

MurΦ proved even more valuable for complex protocols[1], such as LCM. The original, hand-written LCM protocol contained numerous bugs that consumed months of effort to fix, and that continually re-emerged as the protocol evolved. MurΦ uncovered approximately 25 errors in the Teapot LCM specification.[2] After verifying the Teapot code for LCM, we ran the automatically generated C code on several applications with little effort. The remaining problem was an error in a support function that was not verified.

Model checking technology will doubtlessly improve and allow larger protocols and systems to be checked. Researchers are exploring techniques that exploit symmetry or domain-specific knowledge [22] to make systems less dependent on a brute-force exploration of a state space. Teapot is poised to benefit from the progress in this area.

8 Related Work

Distributed shared memory (DSM) systems are an active area of research since Li's first system [20]. Most systems focus on a single general-purpose protocol that, hopefully, is efficient for a wide range of programs. Munin [5] was the first DSM system to support a limited collection of protocols intended for different sharing patterns. Recent systems [17, 23] take a different approach and expose the primitives necessary to implement a coherence protocol. Distributed object systems [3, 6, 16] also provide primitives to support different object coherence protocols. Teapot is not tied to a particular system and could be used with any of them.

Our work most closely resembles the PCS system by Uehara et al. at the University of Tokyo [25]. They described a framework for writing coherence protocols for distributed file system caches. Unlike Teapot, they use an interpreted language (implemented on Tcl!). 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 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 MurΦ verification system.

Wallach et al. propose Optimistic Active Messages [26] that permit the use of blocking primitives inside handlers. They detect at runtime whether a handler involves a blocking primitive; and if so, they launch a separate thread in which to rerun the handler.

Synchronous programming languages, such as ESTEREL [4], are useful for describing reactive systems and real-time applications. Teapot resembles ESTEREL in that it provides a specification of the control part of the protocol, leaving data manipulation to separately written (often in C) support routines. Like ESTEREL, Teapot supports verification and can be translated to executable code. Teapot differs from ESTEREL in that its emphasis is on simplifying the task of programming complicated finite-state machines.

Continuations can express coroutines [13] and parallelism [12, 27]. However, few domain-specific languages exploit continuations, perhaps because of concerns about their implementation complexity and cost. Teapot demonstrates that a restricted form of this feature can be implemented easily and efficiently, without losing its benefits.

Draves et al. [10] used continuations to implement thread management and communication in an operating system. They found many benefits, including reducing the number of kernel stacks from one per thread to one per processor, and unifying implementations of diverse control transfer operations, such as exception handling, preemptive scheduling, and user-level page faults.

The networking community has developed a number of approaches to validating protocols. Besides temporal logic, they also use model-checking techniques based on state-space exploration [15, 21]. To the best of our knowledge,

1. MurΦ simulating LCM had hundreds of times as many configurations as when simulating Stache.

2. With the limited memory available, we could only verify LCM with 2 processor nodes, 1 address, and maximum network reordering of one. Verification of either 2 addresses or more network reordering did not complete, although MurΦ did not report new errors for as long as it ran.

most of their programming models are based on state machines and do not use continuations.

Wing et al. [28] 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.

9 Conclusion

Many programming language features are developed and explored in general-purpose programming languages and rarely find their way into domain-specific languages. This paper provides a counter-example by showing how ideas such as continuations can flow back into a special-purpose language that supports the process of writing and verifying memory-system coherence protocols. These protocols are important to the programming languages community because they facilitate parallel programming and provide an efficient basis for implementing languages and compiler run-time systems. For more information about Teapot, please visit the URL http://www.cs.wisc.edu/~wwt/teapot.

Acknowledgements

Bob Zak of Sun Microsystems pointed us towards automatic verification. Anne Rogers and an anonymous reviewer provided detailed comments that helped improve our presentation. Shubu Mukherjee and Steve Reinhardt generously assisted us with the simulator used in Section 6.

References

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: A Language for Distributed Programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[4] G'erard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. Technical Report nn, Ecole Nationale Sup'erieure des Mines de Paris, nd.

[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[6] S. Chakrabarti, E. Deprit, E.-J. Im, A. Krishnamurthy, C.-P. Wen, and K. Yelick. Multipol: A Distributed Data Structure Library. Technical Report UCB/CSD-95-879, Computer Science Division (EECS), University of California at Berkeley, July 1995.

[7] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 61–75, October 1994.

[8] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[9] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[10] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 122–136, October 1991.

[11] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[12] Christopher T. Haynes and Daniel P. Friedman. Engines Build Process Abstractions. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 18–24, August 1984.

[13] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and Coroutines. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 293–298, August 1984.

[14] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[15] Gerard Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[16] Kirk L. Johnson, M. Frank Kaashoek, and Deborah A. Wallach. CRL: High Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.

[17] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[18] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–218, October 1994.

[19] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[20] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[21] Chung-Shyan Liu. An Object-Based Approach to Protocol Software Implementation. In *SIGCOMM '94*, pages 307–316, London, England, August 1994.

[22] Fong Pong and Michel Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, August 1995.

[23] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[24] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[25] Keiraro Uehara, Hajime Miyazawa, Kouhei Yamamoto, Shigeka-

zu Inohara, and Takasha Masuda. A Framework for Customizing Coherence Protocols of Distributed File Caches in Lucas File System. Technical Report 94-14, Department of Information Science, University of Tokyo, December 1994.

[26] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic Active Messages; A Mechanisms for Scheduling Communication with Computation. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 217–226, August 1995.

[27] Mitchell Wand. Continuation-Based Multiprocessing. In *Conference Record of the 1980 LISP Conference*, pages 19–28, Stanford Univ., Palo Alto CA, August 1980.

[28] Jeannette M. Wing and Mandana Vaziri-Farahani. Model Checking Software Systems: A Case Study. In *Proceedings ACM SIGSOFT Symposium On The Foundations Of Software Engineering*, October 1995.

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* **(** $sub\text{-}args_{opt}$ **) :** *id* **;**
  procedure *id* **(** $sub\text{-}args_{opt}$ **) ;**
protocol:
  protocol *id* **begin** $prot\text{-}decls_{opt}$ **end ;**
prot-decls:
  [*prot-decl*]+
prot-decl:
  var *id* **:** *id* **;**
  const *id* **:=** *id* **;**
  state *id* **(** $state\text{-}args_{opt}$ **)** $\mathbf{transient}_{opt}$ **;**
  message *id* **;**
states:
  [*state*]+
state:
  state *id* **.** *id* **(** $state\text{-}args_{opt}$ **) begin** *msgs* **end ;**
state-args:
  state-arg [**;** *state-arg*]+
state-arg:
  vars **:** *id*
msgs:
  [*msg*]+
msg:
  message *id* **(** $sub\text{-}args_{opt}$ **)** $block\text{-}decls_{opt}$ **begin** *stmts* **end ;**
sub-args:
  sub-arg [**;** *sub-arg*]+
sub-arg:
  var *vars* **:** *id*
  vars **:** *id*
block-decls:
  var [*var-decl*]+
var-decl:
  vars **:** *id* **;**
vars:
  id [**,** *id*]*
stmts:
  ε
  stmt **;** *stmts*
stmt:
  if (*expr* **) then** *stmts* **else** *stmts* **endif**
  if (*expr* **) then** *stmts* **endif**
  while (*expr* **) do** *stmts* **end**
  id **(** *exprs* **)**
  id **:=** *expr*
  suspend (*id* **,** *stmt* **)**
  resume (*id* **)**
  return *expr*
  return
  print (*exprs* **)**
exprs:
  ε
  expr [**;** *expr*]*
expr:
  expr sym-id app-expr
  app-expr
app-expr:
  id **(** *exprs* **)**
  id **{** *exprs* **}**
  atomic-expr
atomic-expr:
  id
  const
  (*expr* **)**