Programming Asynchronous Layers with CLARITY*

Prakash Chandrasekaran
Chennai Mathematical Institute
prakash@cmi.ac.in

Christopher L. Conway
New York University
cconway@cs.nyu.edu

Joseph M. Joy
Microsoft Research India
josephj@microsoft.com

Sriram K. Rajamani
Microsoft Research India
sriram@microsoft.com

ABSTRACT

Asynchronous systems components are hard to write, hard to reason about, and (not coincidentally) hard to mechanically verify. In order to achieve high performance, asynchronous code is often written in an event-driven style that introduces non-sequential control flow and persistent heap data to track pending operations. As a result, existing sequential verification and static analysis tools are ineffective on event-driven code.

We describe CLARITY, a programming language that enables analyzable design of asynchronous components. CLARITY has three novel features: (1) *Nonblocking* function calls which allow event-driven code to be written in a sequential style. If a blocking statement is encountered during the execution of such a call, the call returns and the remainder of the operation is automatically queued for later execution. (2) *Coords*, a set of high-level coordination primitives, which encapsulate common interactions between asynchronous components and make high-level coordination protocols explicit. (3) *Linearity annotations*, which delegate coord protocol obligations to exactly one thread at each asynchronous function call, transforming a concurrent analysis problem into a sequential one.

We demonstrate how these language features enable both a more intuitive expression of program logic and more effective program analysis—most checking is done using simple sequential analysis. We describe our experience in developing a network device driver with CLARITY. We are able to mechanically verify several properties of the CLARITY driver that are beyond the reach of current analysis technology applied to equivalent C code.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Reliability, Verification

Keywords

Concurrency, Asynchronous components, Event-driven programming, Static analysis, Design for analyzability

*An expanded version of this paper is available as a technical report [10].

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

1. INTRODUCTION

High-performance systems components are often written using asynchronous layers. Rather than waiting for a time consuming operation to complete, a component typically executes whatever portion of the operation it can without blocking, records the progress of the operation, and returns to the caller with status "pending." The remainder of the operation executes at a later time. When the operation is finished—perhaps after being blocked and resumed in several thread contexts—a callback function signals completion.

This kind of asynchronous systems programming is usually done in an *event-driven* style which is tailored for performance rather than analyzability: the stages of an operation are "manually scheduled," often by placing them in several different functions which are unrelated in the call graph; asynchronous operations achieve low synchronization overhead using low-level primitives like locks, semaphores, and completion ports; component state is managed manually using heap-allocated data structures like queues. This style of programming leads to efficient implementations, but is difficult and error prone.

Recently, there has been progress in using static analysis tools for error detection [13, 4, 18, 11]. These tools can perform scalable whole-program inter-procedural analysis for sequential programs on properties that do not involve reasoning about the heap, such as locking discipline and the safe initialization and de-allocation of pointers from the stack. Once an object is put into a heap data structure, such as a linked list or queue, these techniques lose precision and become ineffective.

Event-driven programs are non-sequential, asynchronous, and maintain state in the heap for most operations. Thus, most current static analysis tools can check only limited properties of such programs. An enormous amount of research effort has gone into improving the precision and scalability of static analysis for concurrent programs and heap

data, but the performance of these analyses continues to be a significant challenge. This paper approaches the problem from another direction: *Can we write event-driven programs differently, so that they become more analyzable?*

We introduce a programming language, CLARITY, which enables analyzable design of asynchronous components. CLARITY has three novel features: nonblocking function calls, high-level coordination primitives, and linearity annotations.

Nonblocking function calls. Traditional programming languages have two types of calls: synchronous and asynchronous. In synchronous calls, the caller blocks until the callee finishes—if the callee has to wait for resources to become available, the caller waits as well. In asynchronous calls (e.g., POSIX `fork/exec`), the call returns immediately and the body of the called function runs in a separate thread. CLARITY introduces a *nonblocking* call, a new type of asynchronous call that is particularly suited for writing event driven programs. In a nonblocking call, the caller blocks *so long as the callee does not*—if the callee executes a blocking statement, the call returns and the remaining part of the computation is automatically queued for later execution.

The behavior of a nonblocking call can be simulated in C by returning a special "pending" value and manually queuing the remainder of the computation. This is the strategy followed by many asynchronous components. CLARITY allows the programmer to write each operation in a sequential fashion and choose between blocking and non-blocking behavior at the call site. The programmer and analysis software can reason about the call as though it is synchronous, while the CLARITY compiler transforms the call into asynchronous, event-driven code that uses queues to track the state of pending operations.

Coords. CLARITY provides a set of high-level coordination primitives, or *coords*, which encapsulate common interactions between asynchronous components; logical operations are defined sequentially, using coords and event-based communication to indicate synchronization requirements. Each coord has a protocol declaration defining the correct usage of its coordination interface. A sequential static analysis ensures that CLARITY code using the coord follows the protocol along all code paths.

Coords can be thought of as an extension of Hoare and Brinch Hansen's monitors [19, 8] to asynchronous programs with nonblocking calls. Protocol information allows coords to be used both to avoid race conditions and to check for deadlocks.

Linearity annotations. Code annotations in CLARITY delegate protocol obligations to exactly one thread at each asynchronous function call, making the behavior of an operation with respect to each coord effectively sequential. Using the coord protocol, a CLARITY program can be analyzed using simple compositional reasoning: first, we can check that the operation follows the protocol, using a purely sequential analysis; then, assuming that all operations follow the protocol, we can verify that the implementation of the coord does not have deadlocks or assertion violations.

These primitives and design decisions make CLARITY programs easier to analyze—we have checked properties of CLARITY programs using the model checking tools SLAM [4] and ZING [2] that cannot be checked using existing techniques directly on event-driven C programs. In addition, we believe that easy mechanical analysis is correlated with easy human comprehension. In our experience, we find that CLARITY programs are far easier to understand than event-driven programs written in C.

```
STATUS SendPacket(Adapter *a, Packet *p) {
  if( a->AdapterState == NicPausing )
    return STATUS_FAILED;
  INC_REF_CNT(a);
  AcquireSpinLock(&a->sendLock);
  if( HW_IS_AVAIL(a->pHwCsr) ) {
    Status = NICSendPacket(a, p);
    ReleaseSpinLock(&a->sendLock);
    CompletePacket(a, p, Status);
    DEC_REF_CNT(a);
  } else {
    Status = STATUS_PENDING;
    ListAddEnd(a->pSendList, p);
    ReleaseSpinLock(&a->sendLock);
  }
  return Status;
}

void DoPendingSend(Adapter *a) {
   assert( HW_IS_AVAIL(a->pHwCsr) );
   AcquireSpinLock(&a->sendLock);
   Packet *p = ListRemoveHead( a->pSendList );
   Status = NICSendPacket(a, p);
   ReleaseSpinLock(&a->sendLock);
   CompletePacket(a, p, Status);
   DEC_REF_CNT(a);
}

STATUS Pause(Adapter *a) {
  if( a->AdapterState == NicPausing )
    return STATUS_FAILED;
  if(REF_CNT(a) == 0) {
    a->AdapterState = NicPaused;
    PauseComplete(a);
    return STATUS_SUCCESS;
  }
  else {
    Status = NicPausing;
    return STATUS_PENDING;
  }
}

void ReleaseBuffers() {
  ...
  if ( a->AdapterState == NicPausing
       && REF_CNT(a) == 0 ) {
     a->AdapterState = NicPaused;
     PauseComplete(a);
  }
  ...
}
```

Figure 1: Sending packets and pausing in a C driver

2. OVERVIEW

We illustrate the difficulties of analyzing event-driven systems code using code snippets from a network miniport driver. The code in Figure 1 is modeled on the E100BEX driver included in the Microsoft Windows Driver Development Kit and demonstrates typical interactions with the Windows network driver API. The code has been abridged and identifiers changed to simplify the presentation.

The function `SendPacket` transmits the packet `p` via the network adapter `a`. The function is able to transmit the packet (by calling `NICSendPacket`) only if the hardware is available; otherwise, it simply adds the packet to the queue `a->pSendList` and returns `STATUS_PENDING`. It is not obvi-

```
STATUS SendPacket(Adapter *a, Packet *p) {
  if( !(a->sendGate->Enter()) )
    return STATUS_FAILED;
  waitfor( HW_IS_AVAIL(a->pHwCsr), [],
           STATUS_PENDING );
  Status = NICSendPacket(a, p);
  CompletePacket(a, p);
  a->sendGate->Exit();
  return Status ;
}

STATUS Pause(Adapter *a) {
  if( !(a->sendGate->Close()) )
    return STATUS_FAILED;
  waitfor( a->sendGate->IsEmpty(), [a->sendGate->e],
           STATUS_PENDING );
  a->AdapterState = NicPaused;
  PauseComplete(a->AdapterHandle);
  return STATUS_SUCCESS;
}
```

Figure 2: Sending packets and pausing in CLARITY

ous what happens to this packet after it has been added to this queue, since there is no control dependency between `SendPacket` and the code that processes the queue. Packets from this queue are removed and transmitted at several places in the driver code—the logical operation "send packet" is "manually scheduled" across several functions. One such example is shown in function `DoPendingSend`.

A property we might want to check is that every non-failing call to `SendPacket` is followed by a matching call to `CompletePacket`. In the `SendPacket` code, this readily holds if the hardware is immediately available (the second `if` branch). If it is not, the situation is more complicated: the packet is put into a queue from which it is retrieved and completed at a later time, in another function. Because of the difficulty of tracking heap objects and non-sequential control flow, sequential error detection tools are unable to check if every packet is completed along all execution paths.

The function `Pause` in Figure 1 demonstrates another difficulty—ad-hoc coordination between asynchronous operations. This function is the entry point for a "pause" operation, which needs to wait until all outstanding sends are finished before calling `PauseComplete`. The driver maintains a reference count, `REF_CNT(a)`, which tracks the number of outstanding sends in progress. In several unrelated places in the code, inside and outside the `Pause` function, the reference count is checked, updated, and `PauseComplete` is called (e.g., in the function `ReleaseBuffers`). Suppose we wish to automatically check that the pause operation coordinates with all the other operations correctly. This is possible only by doing a global analysis that considers all possible interleavings between pause and other operations, taking into account all of the implicit control dependencies, the reference counts, and the heap objects involved. Such a check is beyond the reach of today's analysis technology.

Figure 2 shows a CLARITY implementation of the send and pause operations. The function `SendPacket` now represents the entirety of the logical operation of sending a packet. Inside `SendPacket`, CLARITY's `waitfor` primitive is used to logically wait until the hardware becomes ready and then transmit the packet. Calls to the `SendPacket` function from the operating system are nonblocking—if the hardware is not ready, the caller is returned the value `STATUS_PENDING` immediately (the third argument to `waitfor`); the remainder of the computation is automatically converted into a closure and put into a queue. The semantics of the CLARITY code are similar to those of the C code in Figure 1, but the programmer does not have to manually schedule the code or manage the persistent state. Moreover, a sequential analysis tool can now easily check that every packet is completed on all execution paths before the `SendPacket` function exits without doing any heap analysis.

```
coord gate
{
  /* Sent when a closed gate is empty. */
  event e;
  /* Called by a "client" thread to enter the gate.
     Returns false if the gate is closed. */
  bool Enter();
  /* Called by a "client" thread to exit the gate.
     If the gate is closed and this is the last thread
     to exit the gate, the event e is sent by Exit(). */
  void Exit();
  /* Called by a "control" thread to close the gate.
     Returns false if the gate is already closed.
     If gate is empty, event e is sent by Close(). */
  bool Close();
  /* Called by a "control" thread waiting
     for the gate to clear. */
  bool IsEmpty();

  protocol{
    enum state {init,s1,s2,done,final} = init;
    Enter.return { /* function exit transition */
      if(state==init && $ret) state = s1;
      elseif (state==init && !$ret) state = done;
      else abort();
    }
    Exit.return { /* function exit transition */
      if(state==s1) state= done;
      else abort();
    }
    Close.return { /* function exit transition */
      if(state==init && $ret) state = s2;
      else if(state==init && !$ret) state = done;
      else abort();
    }
    waitfor { /* invocation transition */
      if(state==s2 && $1 =~ `IsEmpty()` && $2 =~ [e])
        state = done;
      else abort();
    }
    ThreadDone { /* thread exit transition */
      if (state!=done && state!=init)  abort();
    }
}
```

Figure 3: Coord for gate

CLARITY uses higher level abstractions called coords to express coordination between different asynchronous operations. The code in Figure 2 uses `sendGate`, an instance of the `gate` coord, the interface of which is given in Figure 3. The interface has four functions: the first two, `Enter` and `Exit`, are used by "client" threads when they begin and end operations that are controlled by the `gate`; the second two, `Close` and `IsEmpty`, are used by "control" threads. `Close` is used to prevent new operations from beginning and `IsEmpty` is used to check whether pending operations have completed. The `gate` coord models the "asynchronous rundown" of a collection of processes—a common pattern in asynchronous systems programming.

In Figure 2, all send operations first call `Enter()` and then call `Exit()` before returning. The function `Pause` calls `Close()`, then waits for `IsEmpty()` to become true and returns. Unlike Figure 1, there is only one place in the code (inside the body of `Pause`) where the pause operation is completed. At runtime, `Pause` may need to wait asynchronously for pending send operations to complete, but the programmer does not have to worry about these details.

Significantly, CLARITY enables the programmer to make the high-level contract between `Pause` and the other operations explicit. Consequently, it is possible to perform simple compositional analysis automatically and check that the coordination has been implemented and used properly.

Each `coord` declaration is required to specify the sequence of calls by which every logical thread accesses the coord. The `protocol` declaration is given as a SLIC property [5]. The protocol declares a set of variables and then defines transitions caused by triggers, e.g., a function call return, the evaluation of a `waitfor` statement, or thread termination. A transition may inspect and update the values of the protocol variables. A call return transition may inspect the return value using the `$ret` variable. A `waitfor` or call transition may inspect the argument list using positional variables `$1`, `$2`, etc. A transition to an error state is represented by a call to `abort`.

In the case of the `gate` coord, the `protocol` declaration (Figure 3) states that the thread either: (1) calls `Enter` first and, if the call returns `true`, then calls `Exit` (a "client" thread), or (2) calls `Close` first and, if the call returns `true`, then waits until `IsEmpty()` returns true (a "control" thread). Using the protocol specification, a `gate` implementation can be compositionally checked for correct concurrent behavior: assuming that threads using the `gate` obey the protocol, we can verify that the `gate` implementation is deadlock-free.

We can check that the `SendPacket` and `Pause` threads in Figure 2 satisfy the protocol for `gate` by using a per-thread sequential analysis. The compositional reasoning in this case is simplistic, since no threads are created dynamically. If new threads are created, compositional analysis of coord protocol conformance becomes more complicated. We make a particular design choice—every coord protocol instance in progress needs to be handed off to exactly one of the two threads at each asynchronous call; the hand-off is specified using linearity annotations. We illustrate this with another example.

Network File Server. Consider the network file server shown in Figure 4. To read and transmit a large file, the file server launches a set of parallel thread, one to read each block of the file. The threads coordinate to send the blocks in sequence over the network. The code creates the "reader" threads using a `fork` call to `read_block`. It uses `fileChute`, an instance of the `chute` coord, to do the necessary synchronization.

The interface for the coord `chute` is shown in Figure 5. The `protocol` declaration specifies that each thread using the `chute` must: first call `Enter`, which returns an integer token `k`; then call `waitfor(IsMyTurn(k),[e])`, where `e` is the event field of the `chute`; and, finally, call `Exit`. This protocol can be understood as a variation of Lamport's bakery algorithm [22] where the thread may enter a non-critical section after "taking a number" (entering the chute).

Unlike `gate`, there is only one correct usage pattern for a `chute`—there is no distinction between "client" and "control"

```
void read(FILE *fp, int n) {
  chute fileChute;
  for(i = 0; i < n; i++) {
    /* Enter the chute before spawning
      thread, to ensure ordering. */
    int token = fileChute.Enter();
    /* The annotation @fileChute in the call below
       indicates that the remainder of the
       protocol in chute fileChute will be
       carried over by the callee */
    fork read_block(fp,i,token,&fileChute)@fileChute;
  }
}

void read_block(FILE *fp, block i, int token,
                chute *fileChute)
{
  FileBlock fb;
  fb = fs_read(fp,i);
  /* Synch before sending block on the network.
     We omit the return value argument, since the
     return type is void. */
  waitfor( fileChute->IsMyTurn(token), [fileChute->e] );
  /* Send and exit. */
  net_send(fb);
  fileChute->Exit();
}
```

Figure 4: Network file server with asynchronous reading and serialized sending

```
coord chute
{
  /* Sent when a thread exits. */
  event e;
  /* Called by a thread to "get on line"
     in the chute. Returns an integer token
     (the thread's "ticket"). */
  int Enter();
  /* Called by a thread to check if it is
     "first in line" given its token. */
  bool IsMyTurn(int);
  /* Called by a thread to exit the chute.
     Sends the event e. */
  void Exit();

  protocol{
   enum state {init,s1,s2,done,final} = init;
   int k;
   Enter.return { /* function exit transition */
     if(state==init) { k = $ret; state = s1; }
     else abort();
   }
   waitfor { /* invocation transition */
     if(state==s1 && $1 =~ `IsMyTurn(k)` && $2 =~ [e])
       state=s2;
     else abort();
   }
   Exit.return { /* function exit transition */
      if(state==s2) state=done;
      else abort();
   }
   ThreadDone { /* thread exit transition */
     if(state!=done && state!=init) abort();
   }
}
```

Figure 5: Coord for chute

```
void read(FILE *fp, int n) {
  chute fileChute;
  read_block(fp,0,n,&fileChute);
}

void read_block(FILE *fp, int i, int max,
                chute *fileChute)
{
  FileBlock fb;
  if( i==max ) return;
  /* Enter the chute before spawning thread, to
     ensure ordering. */
  int token = fileChute->Enter();
  /* parallel call to the next file block reader. */
  fork read_block(fp,i+1,max,fileChute);
  /* asynchronous part, can execute without any
     ordering */
  fb = fs_read(fp,i);
  /* Synch before sending block on the network. */
  waitfor( fileChute->IsMyTurn(token), [fileChute->e] );
  /* Send and exit. */
  net_send(fb);
  fileChute->Exit();
}
```

Figure 6: Alternate, recursive network file server

threads. Note that `Exit` does not take a token argument—the protocol forbids any thread to call `Exit` except when `IsMyTurn` returns true. Note also that the protocol forbids a thread from trying to "spoof" a token and steal a turn—for each thread the argument to `IsMyTurn` must match the return value of `Enter`.

Note that the thread executing `read` in Figure 4 calls `Enter`, but never calls `IsMyTurn` or `Exit`; likewise, each `read_block` thread calls `IsMyTurn` and `Exit` without first calling `Enter`. How, then, can we verify the `chute` protocol using sequential reasoning?

Whenever a logical thread makes an `fork` call, it effectively creates two logical threads of execution. We require that each coordination protocol in progress be handed off to exactly one of the two threads; each `fork` call is annotated with those instances of the protocol that will be handled by the callee (i.e., the new thread). Note that the `fork` call to `read_block` in Figure 4 is annotated with `@fileChute`. The annotation indicates that the callee, `read_block`, is responsible for completing the protocol for `fileChute`.

Since exactly one logical thread is responsible for carrying out the remainder of the protocol at every asynchronous call, the sequential analysis merely follows one of the two continuations at the call and ignores the other, depending on which instance of the protocol is currently being analyzed (see Section 5).

Recursive Network File Server. Our final example is a recursive implementation of the network file server shown in Figure 6. Instead of generating "readers" from a "master" thread, the recursive implementation has a chain of recursive `fork` calls to `read_block`; each new "reader" thread spawns its own successor.

Note that the calls to `Enter` and `Exit` now always happen in the same thread context. Note also that the `fork` call to function `read_block` in Figure 6 does *not* contain the annotation `@fileChute`. This indicates that the calling thread continues to be responsible for the protocol on `fileChute`.

Again, checking if each thread follows the protocol can be done using purely sequential analysis, one thread at a time. Separately, the correctness of the chute implementation can be established once and for all, assuming that all the client threads conform to the protocol.

3. RELATED WORK

The merits of the event-driven programming style have been the subject of controversy for decades (e.g., [24, 31, 23, 37]). Recent work, e.g., the Capriccio project [38] and Adya et al [1], has focused on capturing the performance of the event-driven style in a more thread-like idiom. Li and Zdancewic have demonstrated how this approach can be incorporated into a language like Haskell [27]. Some of the techniques presented in the above papers (e.g., [37, 38]) could be used to optimize the CLARITY compiler and runtime. However, none of the above efforts address inter-operation coordination in a way that allows for simple compositional reasoning.

The Message Passing Interface (MPI) is a widely used event-driven API for parallel computing [28]. Siegel and Avrunin [34] describe techniques for model checking MPI programs which we believe could be applicable to verifying coord implementations. Strout et al. [36] formulate a data-flow analysis framework for MPI programs. Our emphasis here is on simplifying the analysis problem for event-driven code.

Lee [26] discusses the difficulties of writing correct concurrent software using the threaded model and calls for the use of design patterns for concurrent computation (cf. [25, 32]). We believe coords are exactly these kinds of design patterns. To our knowledge, patterns like `gate` and `chute` have not previously been described in the literature. The coords we present here are inspired by a concurrency library developed by one of the authors—they are design patterns derived from the folk wisdom of systems programmers. CLARITY is an attempt to give language-level support to these abstractions.

The language primitives of CLARITY used for sending and waiting for events are derived from Hoare and Brinch Hansen's monitors [19, 8] and from process calculi such as CCS [29], CSP [20], and the π-calculus [30]. The distinctive feature of CLARITY is the compositional analysis enabled by protocol specifications on coords and the linear hand-off at asynchronous calls.

Coord protocols are similar to De Alfaro and Henzinger's interface automata [12], but are restricted to describing only the input constraints of a single component. Halbwachs et al. [16], Erlingsson and Schneider [14], and Sekar et al. [33] describe protocol enforcement through runtime monitoring. Coord protocols are intended to provide purely static checking.

Simpler programming models for concurrency have been tried before in specialized domains. In the hardware domain, synchronous programming languages like Esterel [6] enforce deterministic concurrency by design and statically schedule the concurrent operations. For cache coherence protocols, Teapot presents a domain specific high-level language that can be both analyzed using model checking and compiled to an implementation [9]. Languages like Cilk [7] and MultiLisp [17] include parallel execution primitives similar to `fork`, but have focused primarily on efficient multiprocessor implementations rather than analyzability.

Stmt	::=	(Send | CallStmt | WaitFor) `;`
Send	::=	(`send` | `sendall`) EventId
CallStmt	::=	Fork | NonBlock | Block
Fork	::=	`fork` CallExpr Annot?
Nonblock	::=	(Lvalue `=`)? `nonblock` CallExpr Annot?
Block	::=	(Lvalue `=`)? `block`? CallExpr
CallExpr	::=	FuncId `(` (CExpr List)? `)`
Annot	::=	`@` ProtocolId List
WaitFor	::=	`waitfor(` WaitCond List (`,` CExpr)? `)`
WaitCond	::=	(LabelId `:`)? CExpr `,` `[` (EventId List)? `]`
A List	::=	A (`,` A)$*$

Figure 7: CLARITY syntax

4. SYNTAX AND SEMANTICS

We give the syntax and semantics for the new language features of CLARITY.

Syntax. CLARITY is an extension of ANSI/ISO C [21]. CLARITY's extensions to C syntax are given in Figure 7. The terminal symbols EventId, FuncId, ProtocolId, and LabelId represent alpha-numeric identifiers with event, function, protocol, and label types, respectively. The terminal Lvalue represents a standard C lvalue expression. The terminal CExpr represents a standard C expression.

A CLARITY Stmt may appear anywhere a statement is allowed in standard C (e.g., in the bodies of loops and `if-then-else` statements). The new statement types are Send, CallStmt, and WaitFor. Send statements (both `send` or `sendall`) use an event identifier. There are three types of call statements: Fork, Nonblock, and Block. Fork and Nonblock calls can take an optional linearity annotation. Block and Nonblock calls can assign their return value to an optional Lvalue. Calls not specified as `fork`, `nonblock`, or `block` are understood to be blocking by default. The WaitFor statement uses an expression (a *return value*) and a (non-empty) list of WaitCond records (*wait conditions*). If the return type of the function in which the statement appears is `void`, the return value may be omitted. A WaitCond record is tagged using an optional *wait label* and uses an expression (the *wait predicate*) and a (possibly empty) list of event identifiers (*wait events*) enclosed in square brackets. The label is used by the runtime to identify the wait condition that enabled execution.

Semantics. We give a partial operational semantics for the new statements and expressions in CLARITY. We omit the semantics for `sendall` and `waitfor` statements with more than one wait condition for space reasons. The full semantics are presented in a technical report [10].

We write $\overline{M}(e)$ for the value of the expression e in memory state M, as defined by the C semantics. We write **false** for the value of the integer constant `0`. We use $\cup$ for set union, $\setminus$ for set difference, $\uplus$ for multiset sum, and $\{\!\{a_1, \ldots, a_n\}\!\}$ for a multiset of elements $a_1, \ldots, a_n$. We elide braces from singleton sets and multisets when the meaning is clear.

Semantic rules are of the form $C \Longrightarrow D$, representing the evolution of the system from configuration C to configuration D. A configuration is a tuple $\langle M, E, Q, P\rangle$ representing a system state with memory state M, set of global events E, multiset of blocked threads Q, and multiset of active threads P. A blocked thread is a tuple, $\langle b, E, S, K\rangle$, representing a thread that has blocked at a `waitfor` statement with wait predicate b, wait events E, next statement S, and continuation stack K. An active thread is a tuple, $\langle S, K\rangle$, representing a thread that is currently executing with next statement S and continuation stack K. A continuation stack is either $\bullet$ (the *empty stack*) or a sequence $k; K$, where k is a continuation and K is a continuation stack. A continuation is either blk `x`.S (a *blocking continuation*) or nbl `x`.S (a *nonblocking continuation*), where `x` is a program variable and S is a program statement (or statement block).

Some of the semantic rules for CLARITY are given in Figure 8. We make several simplifying assumptions. First, since CLARITY statements require only trivial intraprocedural control flow, we assume that each statement is of the form $S_1; S_2$, where S_1 is a CLARITY statement and S_2 is an arbitrary C statement. Second, we treat functions as if they have no arguments. Function arguments can be handled as assignments from actuals to formals; we assume that rules not shown have evaluated these assignments, leaving only the function invocation. Finally, we assume that rules not shown reduce the arguments to `return`, `send`, and `waitfor` from syntactic expressions to values, as necessary: we write `return` v, `send` e, and `waitfor` r b E, where v and r are arbitrary values, e is an event, b is a boolean expression, and E is a set of events (the pair (b, E) represents a single unlabeled wait condition). Rules for C language statements not given are as in ANSI/ISO C.

A `fork` call creates a new running thread descriptor and invokes the called function (CALL-FORK). A *blocking* call adds a blocking continuation (blk) to the stack (CALL-BLK). A *nonblocking* call adds a nonblocking continuation (nbl) to the stack (CALL-NBL). Once the stack has been updated, a called function `f` is expanded into the statement representing its body (CALL). The behavior of the return statement is independent of whether the stack has a blocking or nonblocking continuation (RETURN-BLK and RETURN-NBL, respectively). When the stack is empty, the thread exits (RETURN-EMPTY). The statement `send` e results in the event e being added to the set of global events (SEND).

The `waitfor` statement does not block if the wait predicate evaluates to true and the wait events are available (WAITFOR-SAT). If the `waitfor` statement blocks, the behavior differs depending on whether or not there is a nbl continuation on the stack. If all continuations on the stack are blk continuations, the next statement and the stack are added to the blocked process list—every function in the call stack is blocked until the wait condition is satisfied (WAITFOR-BLK). If there is a nbl continuation on the stack, the next statement and portion of the stack preceding the nbl continuation (the *blocking prefix*) are added to the blocked process list, but the return value argument to `waitfor` is passed to the nbl continuation and the nonblocking caller remains active—control returns to the most recent non-blocking context (WAITFOR-NBL). Note that the return type of all of the functions in the blocking prefix must match—this can be checked using a simple type analysis. When the wait condition of a blocked thread descriptor is satisfied, the thread consumes its wait events and moves from blocked to running (UNBLOCK).

When an external (i.e., non-CLARITY) caller invokes a CLARITY function `f`, a new thread is created for `f` (CALL-EXT). When the thread blocks or exits, the caller receives a return value, as if the call was nonblocking.

The semantics are nondeterministic—if a configuration matches the left-hand side of more than one semantic rule, the system may evolve according to any one of the matched rules. Semantic rules are evaluated atomically. Although

$$\langle M, E, Q, P \uplus \langle \texttt{fork f()}; S, K\rangle\rangle \Longrightarrow \langle M, E, Q, P \uplus \{\{\langle S, K\rangle, \langle \texttt{f()}, \bullet\rangle\}\}\rangle \quad \text{(Call-Fork)}$$

$$\langle M, E, Q, P \uplus \langle \texttt{x = block f()}; S, K\rangle\rangle \Longrightarrow \langle M, E, Q, P \uplus \langle \texttt{f()}, (\texttt{blk x}.S); K\rangle\rangle \quad \text{(Call-Blk)}$$

$$\langle M, E, Q, P \uplus \langle \texttt{x = nonblock f()}; S, K\rangle\rangle \Longrightarrow \langle M, E, Q, P \uplus \langle \texttt{f()}, (\texttt{nbl x}.S); K\rangle\rangle \quad \text{(Call-Nbl)}$$

$$\langle M, E, Q, P \uplus \langle \texttt{f()}, K\rangle\rangle \Longrightarrow \langle M, E, Q, P \uplus \langle S, K\rangle\rangle, \text{ where } S \text{ is the body of } \texttt{f} \quad \text{(Call)}$$

$$\langle M, E, Q, P \uplus \langle \texttt{return } v, (\texttt{blk x}.S); K\rangle\rangle \Longrightarrow \langle M, E, Q, P \uplus \langle \texttt{x = } v; S, K\rangle\rangle \quad \text{(Return-Blk)}$$

$$\langle M, E, Q, P \uplus \langle \texttt{return } v, (\texttt{nbl x}.S); K\rangle\rangle \Longrightarrow \langle M, E, Q, P \uplus \langle \texttt{x = } v; S, K\rangle\rangle \quad \text{(Return-Nbl)}$$

$$\langle M, E, Q, P \uplus \langle \texttt{return } v, \bullet\rangle\rangle \Longrightarrow \langle M, E, Q, P\rangle \quad \text{(Return-Empty)}$$

$$\langle M, E, Q, P \uplus \langle \texttt{send } e; S, K\rangle\rangle \Longrightarrow \langle M, E \cup \{e\}, Q, P \uplus \langle S, K\rangle\rangle \quad \text{(Send)}$$

$$\langle M, E_1 \cup E_2, Q, P \uplus \langle \texttt{waitfor } r\ b\ E_2; S, K\rangle\rangle, \text{when } \overline{M}(b) \neq \textbf{false} \Longrightarrow \langle M, E_1 \backslash E_2, Q, P \uplus \langle S, K\rangle\rangle \quad \text{(WaitFor-Sat)}$$

$$\langle M, E_1, Q, P \uplus \langle \texttt{waitfor } r\ b\ E_2; S, k_1; \ldots; k_n; \bullet\rangle\rangle, \quad \text{(WaitFor-Blk)}$$
$$\text{when } k_i = \texttt{blk x}_i.S_i' \text{ for } 1 \leq i \leq n \text{ and } (\overline{M}(b) = \textbf{false} \text{ or } E_2 \not\subseteq E_1) \Longrightarrow \langle M, E_1, Q \uplus \langle b, E_2, S, k_1; \ldots; k_n; \bullet\rangle, P\rangle$$

$$\langle M, E_1, Q, P \uplus \langle \texttt{waitfor } r\ E_2\ b; S_1, k_1; \ldots; k_n; (\texttt{nbl x}.S_2); K\rangle\rangle, \quad \text{(WaitFor-Nbl)}$$
$$\text{when } k_i = \texttt{blk x}_i.S_i' \text{ for } 1 \leq i \leq n \text{ and } (\overline{M}(b) = \textbf{false} \text{ or } E_2 \not\subseteq E_1) \Longrightarrow \langle M, E_1, Q \uplus \langle b, E_2, S_1, k_1; \ldots; k_n; \bullet\rangle, P \uplus \langle \texttt{x = } r; S_2, K\rangle\rangle$$

$$\langle M, E_1 \cup E_2, Q \uplus \langle b, E_2, S, K\rangle, P\rangle, \text{when } \overline{M}(b) \neq \textbf{false} \Longrightarrow \langle M, E_1 \backslash E_2, Q, P \uplus \langle S, K\rangle\rangle \quad \text{(Unblock)}$$

$$\langle M, E, Q, P\rangle, \text{when } \texttt{f} \text{ is called externally} \Longrightarrow \langle M, E, Q, P \uplus \langle \texttt{f()}, \bullet\rangle\rangle \quad \text{(Call-Ext)}$$

Figure 8: Semantic rules for CLARITY programs.

more than one process may execute in parallel, the set of global events and the blocked and active thread lists will remain consistent. However, the memory state component of a configuration is shared between processes: race conditions can occur if processes access the same location without using a safe coordination scheme.

We assume that the thread scheduler is fair, i.e., that a blocked thread whose wait condition is infinitely often satisfied will eventually move to the active thread list (by application of UNBLOCK) and that every active thread will eventually execute (by evaluation of its next statement). Note that this does not preclude threads blocking indefinitely: there is no guarantee that a wait condition will ever be satisfied (or, indeed, is satisfiable). It is up to the programmer to design the CLARITY program in such a way that deadlock is avoided and wait conditions are eventually satisfied. The use of coords and CLARITY's static analysis can help avoid many concurrency errors.

5. STATIC ANALYSIS

The primary goal of CLARITY's static analysis is to check if coords are implemented and used correctly. We want to check that assertions in the implementation of the coord never fail during execution and that no deadlocks can occur due to the use of coords (i.e., no thread waits for an event that is never sent). One way to verify this is to run a model checker on all of the threads together with the coord implementation and explore the states that arise from all possible interleavings. This approach scales poorly. We exploit the protocol specifications of coords to do compositional analysis: (1) Using sequential analysis (ignoring concurrency), we use the SLAM tool [4] to check that each thread of execution uses coords according to each coord's protocol; (2) Assuming that each thread obeys the coord's protocol, we use the ZING model checker [2] to check that the implementation of the coord is correct.

5.1 Sequential analysis

Coord protocol declarations are transformed into SLIC safety properties for input to SLAM. Recall that we require each coordination protocol in progress to be handed off to exactly one of the two threads at each `fork` call site. This enables the static analysis to transform a CLARITY program with annotations at the `fork` calls to a nondeterministic sequential program. The transformation merely picks one of the two continuations at each parallel call depending on which protocol is currently being analyzed.

This transformation assumes that linearity annotations are consistent with the code. We assume that the programmer does not continue to use a coord after a hand-off to another thread, either explicitly or through an alias. We can use existing techniques to enforce linearity [35, 15].

In sequential type-state analyzers such as SLAM, a type-state property is checked independently on every statically identifiable distinct instance of the given type. There is an internal variable called `curfsm` that holds the current instance being checked. `curfsm` is equal to `NULL` until an instance is detected, e.g., at a variable declaration.

We transform a CLARITY program P to a sequential program $C(P)$ such that we can analyze $C(P)$ instead of P for conformance to the protocol specification φ. The transformation syntactically translates every call `fork foo(args)@`$\texttt{c}_1, \ldots, \texttt{c}_n$ to the program segment shown in Figure 9. We use `if(*)` to represent a nondeterministic choice. In the `if` branch, the `assume` statement allows the analysis to proceed only if `curfsm` is `NULL` or `curfsm` is equal to one of the annotated values $\texttt{c}_1, \texttt{c}_2, \ldots, \texttt{c}_n$. Note that the call to `foo` is a regular sequential call in the transformed program and not a `fork` call. After the call returns, the statement `assume(false)` forces the analysis to stop. In the `else` branch, the `assume` statement allows the analysis to proceed only if `curfsm` is `NULL` or `curfsm` is not equal to any of the values $\texttt{c}_1, \texttt{c}_2, \ldots, \texttt{c}_n$.

We explain this transformation by considering three cases:

1. Suppose `curfsm` is equal to one of the annotated values, say $\texttt{c}_1$. This means that the protocol obligations should be satisfied by the callee. First, consider the `if` branch. Here, the assume statement evaluates to `true`. Thus the analysis proceeds to the call to `foo`. After executing a synchronous call to `foo`, the transformed

```
if(*) {
  assume( curfsm == NULL ∨ [⋁_{1≤i≤n} curfsm == c_i] );
  foo(args);
  ThreadDone();
  assume (false);
} else {
  assume( curfsm == NULL ∨ [⋀_{1≤i≤n} curfsm != c_i] );
}
```

Figure 9: Transformation for a parallel call `fork foo(args)@c1,...,cn`.

code calls `ThreadDone()`, followed by `assume(false)`. Thus the function `foo` is responsible for carrying out the remainder of the protocol on `curfsm`. Next consider the `else` branch. Since `curfsm == c1`, the assume statement evaluates to `false`. Thus further analysis along this path is stopped.

2. Suppose `curfsm` is not null, and not equal to any of the annotated values $c_1, c_2, \ldots, c_n$. This means that the protocol obligations should be satisfied by the caller. In the `if` branch, the `assume` statement evaluates to `false`, stopping the analysis. In the `else` branch, the `assume` statement evaluates to `true` and the remaining code is responsible for carrying out the protocol on `curfsm`.

3. Suppose `curfsm` is null. Then the assume statements in both the `if` and `else` branches evaluate to `true`. Thus, `foo` or the remainder of the callee may initiate a new protocol, and the analysis can track these. (Note that any protocol initiated inside `foo` must also complete before `foo` returns.)

We omit a similar transformation for nonblocking calls. We present the full details of both transformations in a technical report [10].

In addition to coords, protocols can be stated on other objects as well. For example, we might want to check the completion property for each packet `p` that is passed to `SendPacket` in the network driver shown in Figure 2. We can check this property also using a sequential analysis, as long as we follow the programming discipline that at each `fork` only one of the continuations is responsible for completing the protocol and use linearity annotations to guide the analysis.

5.2 Concurrency analysis

The objective of the concurrency analysis is to check the implementation of the coords. We assume that each thread obeys the protocol specified by the coord and use the concurrency-aware model checker ZING to check if the implementation of the coord works correctly under these assumptions.

We automatically convert the protocol specification of the coord to generate a nondeterministic thread that exercises the coord implementation in ways that are allowed by the protocol. Then, we launch a number of these threads in parallel and check the implementation for errors (assertion violations and deadlocks) using ZING.

The checks we describe here prove that the implementation of the coord is correct only with a fixed number of threads. A more general proof is possible, e.g., using parameterized verification [3].

5.3 Guarantees and limitations

Our analysis offers the following guarantee.

THEOREM 1. Consider any CLARITY program P with one coord c. Let φ denote the protocol for c. Suppose each of the threads in the transformed program $C(P)$ satisfies the property φ using sequential analysis (as described in Section 5.1) and the implementation of the coord c satisfies the concurrency analysis check (as described in Section 5.2). Then, during execution of the concurrent program P, there are guaranteed to be no assertion violations in the implementation of c and if a thread in P waits for an event e associated with the coord c, then some thread is guaranteed to send e before exiting.

The proof is presented in a technical report [10].

Our static analysis has two main limitations. The first limitation is that it can detect deadlocks only in programs that use coords for synchronization, and then only for coords used independently. If the programmer uses low-level synchronization primitives or multiple coords in the same block of code, the order in which each thread does blocking `waitfor` operations can result in deadlocks that we will not detect. The second limitation is that we only check safety properties. Thus, if a thread t_1 is waiting for an event through a coord and thread t_2 is obligated to send the event, we can say only that along all code paths, before t_2 exits, the event is indeed sent. We cannot guarantee that t_2 exits and thus we cannot guarantee that the event *will* be sent.

6. IMPLEMENTATION

We wish to demonstrate the viability of our approach in building asynchronous system components with realistic levels of complexity using CLARITY. Along these lines, we have implemented prototype CLARITY development tools and a CLARITY driver for a simple network card, which we have tested in an emulated environment.

Compiler and runtime. The CLARITY compiler transforms a CLARITY source program into C target code. The `send`, `sendall`, and `fork` primitives can be implemented as calls into a CLARITY coordination library. However, translation of the `waitfor` primitive requires more extensive compiler support—if a thread blocks, the CLARITY runtime must be able to restart the thread at a later time, perhaps in the context of a different physical thread, with all of its local state preserved. The compilation uses continuation passing style (CPS) transformations. More details can be found in in a technical report [10].

Device driver implementation. We have written a network device driver in CLARITY for an emulated device we call TINYNIC, comprising about 1,300 lines of code. The target C code produced by the CLARITY compiler is about 2,500 lines.

TINYNIC is closely modeled after hardware such as the Intel E100 network card. We have preserved many of the sources of concurrency and asynchrony, as well as some defining features and idiosyncrasies of network hardware, such as maskable interrupts, memory mapped device registers, and reads and writes via shared memory buffers. We have eliminated most other features that are irrelevant with respect to concurrent and asynchronous behavior (e.g., TINYNIC does not support multicast address filters). We have a software imple-

mentation of the TINYNIC hardware specification that supports concurrent behavior.

Static analysis. We were able to establish properties of the TINYNIC driver by transforming it as described in Section 5.1 and running a sequential analysis on the transformed program.

Our TINYNIC driver uses a `gchute` coord that combines the properties of both the gate and the chute—as in Figure 2, we support the asynchronous rundown of sending packets, but we also use the `chute` protocol to ensure that packets are transmitted in the same order they were submitted. All packets call `Enter` and `Exit` on the `gchute`. The `pause` code closes the gate and waits for all pending sends to complete. The `sendpacket` code uses `waitfor` to wait until the hardware becomes available and uses the `gchute` to enforce packet ordering.

The results for two properties are shown in Table 1. The first property is the protocol for the `gchute`, which is a mixture of both the gate and chute protocols. SLAM was able to check the property on the the transformed CLARITY program in 17.77 seconds, after 9 iterations of iterative refinement, introducing 25 predicates. The second property "packet completion" states that for every packet passed to `SendPacket`, the code gets the size of the packet, transmits at least one fragment of the packet, and calls `CompletePacket`. (Note that this is a weak notion of correctness for `SendPacket` in that we do not require every fragment of the packet to be transmitted, nor that the data transmitted match the packet data.) SLAM was able to check this property on the transformed CLARITY program in 6.82 seconds, after 2 iterations of iterative refinement, introducing 5 predicates.

	CLARITY driver			
Property	Time(s)	Iters	Preds	Result
gchute protocol	17.77	9	25	PASS
packet completion	6.82	2	5	PASS
	C driver			
	Time(s)	Iters	Preds	Result
gchute protocol	*	*	*	*
packet completion	*	*	*	*

Table 1: Sequential checking results

For both properties, SLAM could not finish checking these directly on a hand-coded event-driven C driver. The C driver puts packets that cannot complete immediately into a queue, implemented as a linked heap structure, making analysis difficult. However, the CLARITY code for the TINYNIC driver does not use any queues (though the CLARITY target code and runtime do). It simply keeps the packet as a local variable in the logical thread and uses `waitfor` to block in case the packet cannot be processed. Thus, using an interprocedural analysis (and without reasoning about heap structures), SLAM is able to prove the two properties on the CLARITY code.

Concurrency analysis. We were able to verify the implementations of gate, chute and gchute on small number of threads as shown in Table 2. The coord protocols were used to automatically derive a nondeterministic thread that uses the coord. We ran the concurrency-aware model checker ZING using partial order reduction. In our `gate` implementation (code not shown), the model checker found the following bug: if the gate is closed (by calling `Close`) when there are no pending client threads that have entered, but not exited, then the subsequent call to `waitfor(IsEmpty(),[e])` deadlocks since there is no client thread to send the event `e`. We were able to fix this bug and verify the modified implementation.

coord	Result	num threads	States explored	Time(s)
gate	PASS	3	9133	1
gate	PASS	5	1165393	74
chute	PASS	3	775	0.4
chute	PASS	5	26431	2
chute	PASS	7	1241923	103
gchute	PASS	3	11458	1
gchute	PASS	5	1827952	119

Table 2: Concurrency checking results

Table 2 shows the number of states explored by the model checker and time taken by the model checker in each case. For the gate and gchute, the numbers in the table were obtained after fixing the bug mentioned above.

Runtime testing. We have built a virtual test environment to provide thorough runtime testing of code generated by the compiler and the CLARITY runtime. Our test environment consists of a virtual network hardware implementation, TINYNIC, and a runtime execution environment, TINYNETAPI. The CLARITY-generated driver processes 15,000 packets per second on a 2GHz single processor Pentium machine. The driver is able to initialize and shutdown, including appropriately initializing and resetting the hardware. It is able to handle concurrent sends and receives.

We have kept track of the bug fixes that were necessary in order to get all tests to pass. We note that *none* of the problems we encountered were concurrency or asynchrony bugs; in most cases they were logical errors reflecting a misunderstanding of the hardware specification.

7. CONCLUSIONS

We have presented CLARITY, a language that allows the development of event-driven programs that can be efficiently checked for violations of safety properties. This analyzability is achieved by a careful combination of three language features: nonblocking calls, coords with protocol specifications, and linearity annotations to delegate protocol obligations to exactly one thread at each asynchronous call. Our emphasis to date has been on correctness; our future work will focus on performance.

8. ACKNOWLEDGMENTS

The authors would like to thank Tom Ball, Stephen Edwards, Patrice Godefroid, Karthik Kambatla, Jim Larus, G. Ramalingam and Bill Thies for comments on earlier drafts of this paper.

9. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Usenix Annual Tech. Conf.*, June 2002.

[2] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model

checking concurrent software. In *CONCUR*, volume 3170 of *LNCS*, 2004. Invited paper.

[3] T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, volume 2102 of *LNCS*, pages 221–234, 2001.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop*, volume 2057 of *LNCS*, 2001.

[5] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research, Jan. 2001.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Programming*, 19(2):87–152, 1992.

[7] R. D. Blumofe, C. F. Joerg, B. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, Santa Barbara, CA, July 1995.

[8] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Trans. Softw. Eng.*, 1(2):199–207, 1975.

[9] S. Chandra, B. Richards, and J. R. Larus. Teapot: A domain-specific language for writing cache coherence protocols. *IEEE Trans. Softw. Eng.*, 25(3):317–333, 1999.

[10] P. Chandrasekaran, C. L. Conway, J. M. Joy, and S. K. Rajamani. Programming asynchronous layers with CLARITY. Technical Report MSR-TR-2007-80, Microsoft Research, 2007.

[11] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–69, 2002.

[12] L. de Alfaro and T. A. Henzinger. Interface automata. In *FSE*, pages 109–120, 2001.

[13] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[14] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. Technical Report 1999-1758, Cornell University, Ithaca, NY, 1999.

[15] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.

[16] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology*, June 1993.

[17] R. H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst.*, 7(4):501–538, October 1985.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[19] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.

[20] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[21] ISO Standard - Programming Languages - C, Dec. 1999. ISO/IEC 9899:1999.

[22] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, Aug. 1974.

[23] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance. Technical Report MSR-TR-2001-39, Microsoft Research, 2001.

[24] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, 1979.

[25] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 2000.

[26] E. A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, January 10 2006.

[27] P. Li and S. Zdancewic. Combining events and threads for scalable network services: Implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI*, pages 189–199, June 2007.

[28] Message Passing Interface Forum. MPI: A message-passing interface standard. `http://www.mpi-forum.org`.

[29] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[30] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical Report LFCS Report 89-85, University of Edinburgh, June 1989.

[31] J. K. Ousterhout. Why threads are a bad idea (for most purposes). In *Usenix Annual Tech. Conf.*, 1996. Invited talk.

[32] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Addison-Wesley, 2000.

[33] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. *Security and Privacy*, page 144, 2001.

[34] S. F. Siegel and G. S. Avrunin. Analysis of MPI programs. Technical Report UM-CS-2003-036, University of Massachusetts, 2004.

[35] F. Smith, D. Walker, and J. G. Morrisett. Alias types. In *ESOP*, volume 1782 of *LNCS*, pages 366–381, 2000.

[36] M. M. Strout, B. Kreaseck, and P. D. Hovland. Data-flow analysis for MPI programs. In *ICPP*, Aug. 2006.

[37] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS*, 2003.

[38] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP*, 2003.