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 Joy
Microsoft Research India
josephj@microsoft.com

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

ABSTRACT

Asynchronous systems programs are usually written in an event-driven style which is tailored for performance rather than analyzability. Such programs have non-sequential control flow and make heavy use of heap data structures to store and retrieve state related to pending operations. As a result, existing tools that analyze sequential programs are ineffective in analyzing asynchronous systems components.

We describe CLARITY, a programming language that enables analyzable design of asynchronous components. CLARITY has three novel features: (1) *Nonblocking* function calls that 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, encapsulate common interactions between asynchronous components and make high-level coordination protocols explicit. (3) *Linearity annotations* 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, testing, and analyzing a network device driver using CLARITY.

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—perhaps in a different thread context. 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, 6, 17, 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 problem. This paper attempts to change the statement of the problem: *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. When a blocking statement is executed

during a nonblocking call, 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" systems interfaces. 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.

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 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. Since ease of understanding is subjective, we will focus on more objective criteria: we demonstrate that we can analyze and check properties in CLARITY programs that cannot be checked using existing techniques directly on event-driven C programs.

2. OVERVIEW

Motivating example. We illustrate the difficulties of analyzing event-driven systems code using code snippets from a network miniport driver (Figure 1).

The function `sendpacket` transmits the packet pointed-to by the function argument `p`. 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`. It is not obvious 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" or "ripped" across several functions. One such example is shown in function `doPendingSend`.

A property we might want to check is that every packet passed to `sendpacket` is "completed" along all code paths (by calling `CompletePacket`). In the `sendpacket` code, this

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

/* Called with sendLock held */
STATUS doPendingSend(Adapter *a) {
   assert( !NIC_IS_IDLE_OR_BUSY(a->pHwCsr) );
   a->pCurPacket = ListRemoveHead( a->pSendList );
   Status = NICSendPacket(a);
   CompletePacket(a, a->pCurPacket);
   DEC_REF_CNT(a);
   return Status;
}

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

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

Figure 1: Sending packets with pausing using C

readily holds in the `if` branch. The situation is more complicated in the `else` branch, since 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.

Often, different operations need to coordinate; such coordination is implemented using ad hoc synchronization mechanisms that are hard to understand and reason about. For example, consider the function `pause` in Figure 1. This function is the entry point for a "pause" operation, which needs to wait until all outstanding sends are finished before it informs the operating system that it has completed (by calling `PauseComplete`). This particular 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. Suppose we wish to automatically check that the

```
STATUS sendpacket(Packet *p, Adapter *a)
{
  if( !(a->sendGate->Enter()) )
    return STATUS_FAILED;
  waitfor( STATUS_PENDING,
           !NIC_IS_IDLE_OR_BUSY(a->pHwCsr), [] );
  Status = NICSendPacket(a);
  CompletePacket(a, p);
  a->sendGate->Exit();
  return Status ;
}
STATUS pause(Adapter *a)
{
  if( !(a->sendGate->Close()) )
    return STATUS_FAILED;
  waitfor( STATUS_PENDING, a->sendGate->IsEmpty(),
           [a->sendGate->e] );
  a->AdapterState = NicPaused;
  PauseComplete(a->AdapterHandle);
  NICDisableInterrupt(a);
  return STATUS_SUCCESS;
}
```

Figure 2: Sending packets with pausing using CLARITY

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 the implicit control dependencies, the reference counts, and the heap objects involved. Such a check is beyond the reach of today's analysis technology.

Miniport driver in CLARITY. Figure 2 shows a CLARITY implementation of the `sendpacket` and `pause` functions.

All the code that handles the logical operation of sending a packet is now present together in `sendpacket`. 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 first argument to `waitfor`); the remainder of the computation is automatically converted into a closure and put into a queue. Thus, it works essentially like the 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.

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. All send operations "enter" the gate first by calling `sendGate->Enter()` and "exit" the gate before returning by calling `sendGate->Exit()`. The function `pause` merely "closes" the gate by calling `sendGate->Close()`, then waits for the gate to become "empty" 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

```
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{
       if(state==init && $ret) state = s1;
       elseif (state==init && !$ret) state = done;
       else abort();
    }
    Exit.return{
       if(state==s1) state= done;
       else abort();
    }
    Close.return{
       if(state==init && $ret) state = s2;
       elseif (state==init && !$ret) state = done;
       else abort();
    }
    waitfor{
      if($1 == IsEmpty() && $2 == e) state = done;
      else abort();
    }
    ThreadDone{
      if (state==done || state==init) state = final;
      else abort();
  }
}
```

Figure 3: Coord for gate

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. Consider again the example from Figure 2. The object `a->sendGate` is an instance of the coord `gate`, whose interface 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. A `gate` can be implemented in a few dozen lines of code using an atomic counter and a boolean flag.

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 [7]. The protocol declares a set of variables and then defines tran-

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

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

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

sitions 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 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 create a deductive proof or run an automated model checker to show 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 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 an example.

File Server. Consider the network file server shown in Figure 4. To read and transmit a large file with `n` blocks, the file server launches `n` parallel threads—one to read each block. After launching the reads in parallel, the main thread waits for the reads to complete and sends them in sequence over the network. The code spawns `n` logical threads each of which execute `read_block`. The spawning is done recursively by the `fork` call to `read_block` inside the body of `read_block`. The code uses `c`, an instance of the `chute` coord, to do the necessary synchronization. We omit the first argument to `waitfor` because the return type of `read_block` is `void`.

```
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 token;
   Enter.return{
      if(state==init){token = $ret; state = s1;}
      else abort();
   }
   waitfor{
       if(state==s1 && $1==IsMyTurn(token) && $2==e)
           state=s2;
       else
           abort();
   }
   Exit.return{
      if(state==s2) state=done;
      else abort();
   }
   ThreadDone{
      if(state==done || state==init) state = final;
      else abort();
   }
}
```

Figure 5: Coord for chute

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`, the return value of which is an integer `token`; then call `waitfor(IsMyTurn(token),e)`, where `e` is the event belonging to the `chute`; finally, call `Exit`. This protocol can be understood as a variation of Lamport's bakery algorithm [22] where the thread may enter an 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" 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`. 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.

Iterative File Server. Our final example is an iterative implementation of the network file server, shown in Figure 6. Instead of a chain of `n` recursive `fork` calls to `read_block`,

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

void read_and_send_block(FILE *fp, block i, int token,
                         chute *c)
{
  FileBlock fb;
  fb = fs_read(fp,i);
  /* Synch before sending block on the network. */
  waitfor( c->IsMyTurn(token), [c->e] );
  /* Send and exit. */
  net_send(fb);
  c->Exit();
}
```

Figure 6: Alternate implementation for Network file server

the iterative implementation has a "master" thread that generates `n` parallel calls to the function `read_and_send_block` inside a loop. Note that the thread executing `read` calls `Enter`, but never calls `IsMyTurn` or `Exit`; likewise, each `read_and_send_block` thread calls `IsMyTurn` and `Exit` without first calling `Enter`.

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_and_send_block` in Figure 6 is annotated with `@c`. The annotation indicates that the callee `read_and_send_block` is responsible for completing the protocol for chute `c`. Note that the recursive `fork` call to function `read_block` in Figure 4 does *not* contain the annotation `@c`. This indicates that the calling thread continues to be responsible for the protocol on `c`.

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).

3. RELATED WORK

The merits of the event-driven programming style have been the subject of controversy for decades (see, e.g., [24, 30, 23, 35]). Recent work, e.g., the Capriccio project [36] 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., [35, 36]) 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.

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, 31]). 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 represent design patterns derived from the folk wisdom of systems programmers. CLARITY is an attempt to capture this folk wisdom and give language-level support to these abstractions. Further, CLARITY allows programmers to write their own coords, which allows the development of customized coordination schemes. with the aid of modular static analysis tools. For example, our TINYNETAPI driver (see Section 6) uses a `gchute` coord which combines elements from both the `gate` and `chute` coords.

The language primitives of CLARITY used for sending and waiting for events are derived from process calculi such as CCS [28], CSP [19], and the π-calculus [29]. Coords have some similarity to Hoare's monitors [18]. The distinctive feature of CLARITY's coords are the protocol specification and the linear hand-off at asynchronous calls, which allows compositional analyzability.

Our compilation strategy relies in part on a transformation to continuation-passing style (CPS) [3], which requires collecting the execution environment of the current function in a form that can be stored and resumed. This is a well-studied problem. Scheme's closures [34] have been the most influential work in this area. The idea has also appeared before in Algol's thunks [20] and in Hewitt's actors [32]. Several systems provide light-weight threads or "fibers", which allow programmers to create, save and resume closures efficiently [35, 36, 1]. These mechanisms can been used to mimic the behavior of CLARITY's blocking primitives, but they do not constitute a full solution to the difficulties of asynchronous programming—because of the lack of high-level coordination patterns like coords, state still needs to be managed manually and kept on the heap, limiting the analyzability of the code.

Demsky [12] and Fischer et al. [15] describe CPS transformations from threaded to event-driven code similar to the one implemented in the CLARITY compiler. Demsky's transformation creates a new continuation at every blocking I/O call and relies on the scheduler to invoke the continuation when the I/O is complete. Fischer et al. introduce a `wait` primitive similar to our `waitfor` and require potentially blocking functions to have a special type modifier. Our key contribution is to move consideration of blocking vs. non-blocking call behavior to the caller and leverage the simpler sequential semantics of the source program to perform precise program analysis.

Simpler programming models for concurrency have been tried before in specialized domains. In the hardware domain, synchronous programming languages like Esterel [8] 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 [10]. Languages like Cilk [9] and Mul-

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(` CExpr `,` WaitCond List `)`
WaitCond	::=	(LabelId `:`)? CExpr `,` `[` EventId List `]`
A List	::=	A (`,` A)$*$ | ϵ

Figure 7: CLARITY syntax

tiLisp [16] include parallel execution primitives similar to `fork`, but have focused primarily on efficient multiprocessor implementations rather than analyzability.

4. SYNTAX AND SEMANTICS

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

Syntax. CLARITY is a superset of standard C [21]. CLARITY's extensions to C syntax are given in Figure 7. The productions EventId, FuncId, ProtocolId, and LabelId represent alpha-numeric identifiers with event, function, protocol, and label types, respectively. The production Lvalue represents a standard C lvalue expression, The production 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 (possibly 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 Appendix A.

Let A be a set. We use 2^A to denote the powerset of A and A^* to denote the set of multisets of elements from A. We use $\cup$ for set union, $\uplus$ for multiset sum, and $\{\!\{a_1, \ldots, a_n\}\!\}$ for a multiset of elements $a_1, \ldots, a_n$.

Let **Var**, **Expr**, **Stmt** be sets of, respectively, *variables*, *expressions*, and *statements* appearing in the program. Let **Locs** be a set of *locations*, **Vals** $\subseteq$ **Expr** a set of *values*, and **Evts** a set of *events*. Let **false** and **true** be elements of **Vals** such that **false** $\neq$ **true**.

Let $\mathcal{M} =$ **Locs** $\rightarrow$ **Vals** be the set of *memory states*, functions from locations to values. Let $\overline{M}$: **Expr** $\rightarrow$ **Vals** be a function from expressions to values in memory state M. We require that $\overline{M}(v) = v$ for all $v \in$ **Vals**.

Let $\mathcal{C}$ be the set of *continuations*. A continuation is either `b|k` `x`.S (a *blocking continuation*) or `nb|` `x`.S (a *nonblocking continuation*), where `x` $\in$ **Var** and $S \in$ **Stmt**. Let $\mathcal{K}$ be the set of *continuation stacks*. 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.

Let $\mathcal{B} =$ **Expr** $\times 2^{\textbf{Evts}} \times$ **Stmt** $\times \mathcal{K}$ be the set of *blocked thread descriptors*. Each $\langle b, E, S, K\rangle \in \mathcal{B}$ represents a thread that has blocked at a `waitfor` statement with wait predicate b, wait events E, next statement S, and continuation stack K.

Let $\mathcal{R} =$ **Stmt** $\times \mathcal{K}$ be the set of *running thread descriptors*. Each $\langle S, K\rangle \in \mathcal{R}$ represents a thread that is currently executing with next statement S and continuation stack K.

Let $\mathcal{S} = \mathcal{M} \times 2^{\textbf{Evts}} \times \mathcal{B}^* \times \mathcal{R}^*$ be the set of *system configurations*. Each $\langle M, E, Q, P\rangle \in \mathcal{S}$ represents a system configuration with memory state M, set of global events E, multiset of blocked thread descriptors Q (the *blocked thread list*), and multiset of running thread descriptors P (the *active thread list*).

Some of the semantic rules for CLARITY are given in Figure 8. Semantic rules are of the form $C \Longrightarrow D$, representing the evolution of the system from configuration C to configuration D. The semantics are nondeterministic—if a configuration C 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 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 make several simplifying assumptions in the semantic rules. 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 where $v \in$ **Vals**; `send` e where $e \in$ **Evts**; and `waitfor` r b E where $r \in$ **Vals**, $b \in$ **Expr**, and $E \subseteq$ **Evts** (the pair (b, E) represents a single unlabeled wait condition). An empty list of wait conditions is equivalent to a single wait condition with the wait predicate **false** (i.e., it is unsatisfiable). The CLARITY compiler will issue a warning in this case.

A `fork` call creates a new running thread descriptor and invokes the called function (CALL-FORK). A *blocking* call adds a blocking continuation (`b|k`) to the continuation stack (CALL-BLK). A *nonblocking* call adds a nonblocking continuation (`nb|`) to the continuation stack (CALL-NBL). Once the continuation 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 continuation stack has a blocking or nonblocking continuation (RETURN-BLK and RETURN-NBL, respectively). When the continuation stack is empty, the

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 8: Semantic rules for CLARITY programs.

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 `nb|` continuation on the stack. If all continuations on the stack are `b|k` continuations, the next statement and the continuation 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 `nb|` continuation on the stack, the next statement and portion of the stack preceding the `nb|` continuation (the *blocking prefix*) are added to the blocked process list, but the return value argument to `waitfor` is passed to the `nb|` continuation and the non-blocking 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 move 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.

Rules for C language statements not given are as in standard C.

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., it is not the case that one thread is waiting for an event that is never sent). One way to check this is to model check all threads together with the states of the coords 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 [6] 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 specifications are SLIC properties that SLAM can accept as input. 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 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. Figure 9 gives examples of such inconsistencies. We can use existing techniques to enforce linearity [33, 14].

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

```
read(FILE *fp, int n, chute *c) {
  for(i = 0; i < n; i++){
    int token = c->Enter();
    //aliasing c2 with c is illegal
    //since c is handed off below
    chute *c2 = c;
    fork read_and_send_block (fp,i,token,c)@c;
    //the following reference to c
    //is illegal since c has been handed off
    waitfor(c->isMyTurn(token),[c->e]);
    //the reference to c through the alias
    //c2 is also illegal
    waitfor(c2->isMyTurn(token),[c2->e]);
  }
}
```

Figure 9: Incorrect usage of "@" annotations

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)@`$c_1, \ldots, c_n$ to the program segment shown in Figure 10. 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 $c_1, c_2, \ldots, c_n$. 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 $c_1, c_2, \ldots, c_n$.

We explain this transformation by considering three cases:

1. Suppose *curfsm* is equal to one of the annotated values, say c_1. This means that the protocol obligations should be satisfied by the callee. First, consider the `if` branch. Here, the condition $\bigvee_{1 \le i \le n}(curfsm = c_i)$ evaluates to `true`. Thus, the analysis proceeds to the call to `foo`. After executing a synchronous call to `foo`, the transformed 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 $curfsm = c_1$, the condition $(curfsm = \textbf{null}) \vee \bigwedge_{1 \le i \le n}(curfsm \neq c_i)$ 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 and the remaining code is responsible for carrying out the remainder of 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.

```
1: if * then
2:    assume curfsm = null ∨ [⋁_{1≤i≤n} curfsm = c_i]
3:    foo(args);
4:    ThreadDone();
5:    assume (false);
6: else
7:    assume curfsm = null ∨ [⋀_{1≤i≤n} curfsm ≠ c_i]
8: end if
```

Figure 10: Transformation for a parallel call `fork foo(args)@`$c_1, \ldots, c_n$.

```
read_block(FILE *fp, int i, int max, chute *filechute) {
  ...
  /* fork call to the next file block reader. */
  if(*){
    assume(curfsm == null);
    read_block(fp, i+1, max, filechute);
    ThreadDone();
    assume(false);
  } else {
    assume(true);
  }
  ...
}
```

Figure 11: Sequential analysis for code in Figure 4

We omit a similar transformation for nonblocking calls. We present the full details of both transformations in Appendix B.

We illustrate this using two examples. First, consider the file server example from Figure 4, where the call to `read_block` is annotated with an empty list. A portion of the transformed sequential program $C(P)$ is shown in Figure 11. Consider the `if` branch first. Since the annotation list is empty the assume statement in the `if` branch passes only if `curfsm == null`, and the assume statement in the `else` branch always passes. Thus, if `curfsm = filechute`, then only the `else` branch is analyzed and the continuation of the `else` branch is responsible for the remainder of the protocol on `filechute`.

Next, consider the alternate implementation of the file server from Figure 6, where the call to `read_and_send_block` is annotated with `@c`. The resulting sequential program is shown in Figure 12. Here the annotation at the `fork` call is the singleton `c`. Thus, the assume statement in the `if` branch passes if `curfsm == null` or `curfsm == c`. Thus, the body of the callee `read_and_send_block` is obligated to carry out the remainder of the protocol on `c`.

In addition to coords, there are other objects on which usage protocols can be stated. 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 given the responsibility 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

```
read(FILE *fp, int n) {
  chute *c = new chute();
  for(int i = 0; i < n; i++) {
    /* Enter the chute before spawning
       thread, to ensure ordering. */
    int token = c->Enter();
    if(*) {
      assume((curfsm == null) || (curfsm == c));
      read_and_send_block(fp,i,token,c);
      ThreadDone();
      assume(false);
    } else {
      assume((curfsm == null) || (curfsm != c));
    }
  }
}
```

Figure 12: Sequential analysis for code in Figure 6

thread obeys the protocol specified by the coord and use a concurrency-aware model checker 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 using our concurrency-aware model checker ZING. Failures here may manifest as assertion violations in the implementation of the coord or as deadlocks.

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 the techniques of parameterized verification [5].

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.

PROOF. (Sketch) Suppose both the sequential analysis and the concurrency analysis pass, and still the program P either fails an assertion inside coord c, or deadlocks on an event in c. Consider the run r that leads to the assertion failure or deadlock. Suppose there is some thread that violates the coord protocol in r. This contradicts the assumption that the sequential analysis has certified all threads as individually obeying the coord protocol. Thus, all threads have to obey the coord protocol for c in r. Now consider the calls made to the coord by all the threads in r. Since every thread satisfies the protocol, such a test should have been exercised by the concurrency analysis, contradicting the assumption that the concurrency analysis passes.

□

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—this is exactly the termination problem—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 a prototype CLARITY compiler, static analysis tools, and runtime 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 Appendix C.

Device driver implementation. We have written a network device driver in CLARITY for a 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, a memory mapped Control/Status Register (CSR) 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 implementation of the TINYNIC hardware specification that supports the concurrent behavior.

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

Note that our methodology allows programmers to write their own coords. As long as the coords come with a protocol specification, we can use our analysis methodology to check both the CLARITY code that uses the coord (using sequential analysis) and the implementation of the coord (using concurrency analysis). 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 to 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 "completes" the packet by calling a completion function. (Note that this is weaker than the full packet completion property.) 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 the event driven C driver. The C driver put 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, but it is unable to prove those directly on the event driven C code.

To better illustrate this difficulty, consider the two simplified code fragments shown in Figures 13 and 14. We want to check that for every packet `p`, first `A(p)` is called, and then `B(p)` is called. The code in Figure 13 illustrates the analysis problem for the CLARITY code. Here the analysis problem presented to SLAM is much simpler—the queue has been abstracted away and the operation can be verified by analyzing sequential control-flow. The code in Figure 14 illustrates the analysis problem for existing event driven code written in C. Here, SLAM needs to analyze the heap, since the packets are queued in a list if the hardware is not available.

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

```
STATUS sendpacket(Packet *p){
  A(p);
  waitfor(STATUS_PENDING,cond,[]);
  B(p);
  return STATUS_SUCCESS;
}
```

Figure 13: An analysis problem for CLARITY code.

```
List *list;
STATUS sendpacket(Packet *p) {
  A(p);
  if(cond) {
    B(p);
    return STATUS_SUCCESS;
  } else {
    ListAddEnd( list, p);
    return STATUS_PENDING;}
}
STATUS HandlePendingSendsInList() {
  if(ListHasElements(list) {
    Packet *p = ListRemoveHead(list );
    B(p);}
  return STATUS_SUCCESS;
}
```

Figure 14: An analysis problem for C code

ZING using partial order reduction. For the `gate` implementation, 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, that serves as a host for the device driver. The environment, implemented with over 10,000 lines of C and C++ code. A block diagram can be found in Figure 15.

TINYNETAPI, the execution environment, implements a subset of the kernel mode network driver interface in our target operating system. As with TINYNIC, most sources of concurrency are preserved—concurrent and asynchronous sending and receiving of packets, interrupt and DPC/tasklet

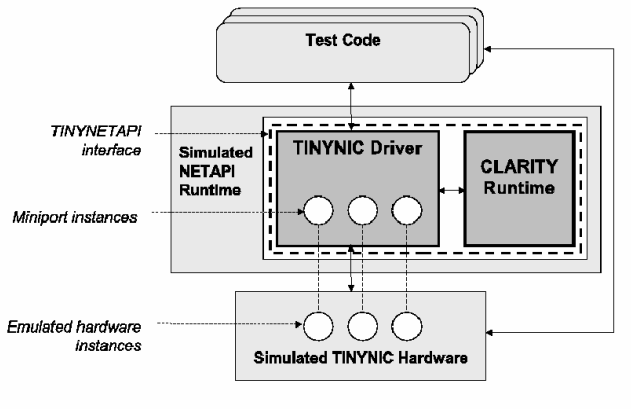

Figure 15: CLARITY **simulation environment**

handling; support for pausing, halting and unloading the driver; and so on. In addition, numerous dynamic checks have been put in place to validate proper driver behavior. For example, the uses of spinlocks and memory allocations are individually tracked. The environment includes utilities that make it easy to write simple test programs which can, for example, submit concurrent streams of packets and pause the driver midstream.

The CLARITY-generated driver processes 15,000 packets per second on a 2GHz single processor Pentium machine. The driver passes the following tests:

1. Ability to initialize and shutdown, including appropriately initializing and resetting the hardware.
2. Ability to handle concurrent sends, pausing the driver midstream.
3. Ability to handle concurrent sends and receives.

We have kept track of the bug fixes we have needed to make in order pass all tests. It is encouraging to note that none of the errors have been issues of concurrency or asynchrony, but rather logical errors with respect to the hardware specification. We have of course had to fix concurrency-related errors in the runtime as it was under development, but none in the CLARITY driver code—we appear to have made some progress toward our goal of simplifying the driver development in the areas of concurrency and asynchrony.

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 has been on the proper functioning of the driver; our future work will be to focus on the performance of generated code and of the runtime.

8. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Usenix 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*, 2004. Invited paper.

[3] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] A. W. Appel and Z. Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[5] T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, 2001.

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

[7] 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.

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

[9] R. D. Blumofe, C. F. Joerg, B. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Symposium on Principles and Practice of Parallel Programming*, July 1995.

[10] 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.

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

[12] B. C. Demsky. An empirical study of technologies to implement servers in Java. Master's thesis, MIT, 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] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.

[15] J. Fischer, R. Majumdar, and T. Millstein. Tasks: Language support for event-driven programming. In *PEPM*, Jan. 2007.

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

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

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

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

[20] P. Z. Ingerman. Thunks—a way of compiling procedure statements with some comments on procedure declaratations. *Commun. ACM*, 4(1):55–58, Jan. 1961.

[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. A language-based approach to unifying events and threads, Apr. 2006. Draft.

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

[29] 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.

[30] J. K. Ousterhout. Why threads are a bad idea (for most purposes). In *Usenix Tech. Conf.*, 1996. Invited talk. `http://home.pacbell.net/ouster/threads.pdf`.

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

[32] B. C. Smith and C. Hewitt. *A PLASMA Primer*. MIT AI Lab,

Cambridge, MA, Oct. 1975. Draft.
[33] F. Smith, D. Walker, and J. G. Morrisett. Alias types. In *ESOP*, 2000.
[34] G. J. Sussman and G. L. Steele, Jr. Scheme: An interpreter for extended lambda calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab, Dec. 1975.
[35] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS*, 2003.
[36] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP*, 2003.

APPENDIX

A. OPERATIONAL SEMANTICS

Let A be a set. We use 2^A to denote the powerset of A and A^* to denote the set of multisets (or bags) of elements from A. We use $\cup$ for set union, $\uplus$ 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.

Let **Var**, **Expr**, **Stmt**, **Lab** be sets of, respectively, *variables*, *expressions*, *statements*, and *wait labels* appearing in the program. **Stmt** includes *compound statements*, i.e., statements of the form $S_1; S_2$ where $S_1, S_2 \in$ **Stmt**.

Let **Locs** be a set of *locations*, **Vals** $\subseteq$ **Expr** a set of *values*, and **Evts** a set of *events*. Let **false** and **true** be elements of **Vals** such that **false** $\neq$ **true**. Let $\mathcal{M} =$ **Locs** $\rightarrow$ **Vals** be a set of *memory states*, functions from locations to values. Let $\overline{M}$: **Expr** $\rightarrow$ **Vals** be a function from expressions to values in memory state $M \in \mathcal{M}$. We require that $\overline{M}(v) = v$ for all $v \in$ **Vals**.

Let $\mathcal{C}$ be a set of *continuations*. A continuation is either `blk` `x`.S (a *blocking continuation*) or `nbl` `x`.S (a *non-blocking continuation*), where `x` $\in$ **Var** and $S \in$ **Stmt**. Let $\mathcal{K}$ be a set of *continuation stacks*. A continuation stack is either $\bullet$ (the *empty stack*) or $k; K$, where k is a continuation and K is a continuation stack.

Let $\mathcal{W} =$ **Lab** $\times$ **Expr** $\times 2^{\textbf{Evts}}$ be a set of *wait conditions*. Each $\langle \ell, b, E \rangle \in \mathcal{W}$ represents a wait condition with label ℓ, wait predicate b, and wait events E.

Let $\mathcal{B} = 2^{\mathcal{W}} \times$ **Stmt** $\times \mathcal{K} \times 2^{\textbf{Evts}}$ be the set of *blocked thread descriptors*. Each $\langle W, S, K, L \rangle \in \mathcal{B}$ represents a thread that has blocked at a `waitfor` statement with wait conditions W, next statement S, continuation stack K, and local events L.

Let $\mathcal{R} =$ **Stmt** $\times \mathcal{K} \times 2^{\textbf{Evts}}$ be a set of *running thread descriptors*. Each $\langle S, K, L \rangle \in \mathcal{R}$ represents a thread that is currently executing with next statement S, continuation stack K, and local events L.

Let $\mathcal{S} = \mathcal{M} \times 2^{\textbf{Evts}} \times \mathcal{B}^* \times \mathcal{R}^*$ be the set of *system configurations*. Each $\langle M, E, Q, P \rangle \in \mathcal{S}$ represents a system configuration with state M, global events E, multiset of blocked thread descriptors Q (the *blocked thread list*), and multiset of running thread descriptors P (the *active thread list*).

We make several simplifying assumptions in our semantics. First, 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. We also assume that rules not shown reduce the arguments to `return`, `send`, and `waitfor` from syntactic expressions to values, as necessary: we write `return` v where $v \in$ **Vals**; `send` e where $e \in$ **Evts**; and `waitfor` r W where $r \in$ **Vals** and $W \subseteq \mathcal{W}$.

The semantic rules for CLARITY are given in Figure 16.

1: **if** * **then**
2: **assume** $((curfsm = \textbf{null}) \vee \bigvee_{1 \leq i \leq n} (curfsm = c_i))$
3: x = foo(args);
4: ThreadDone();
5: **assume** (false);
6: **else**
7: x = **nondet**$(v_1, v_2, \ldots, v_m)$;
8: **assume** $((curfsm = \textbf{null}) \vee \bigwedge_{1 \leq i \leq n} (curfsm \neq c_i))$
9: **end if**

Figure 17: Transformation for a `nonblock` call `x = nonblock foo(args)`.

Rules for C language statements not given are as in standard C.

B. SEQUENTIAL ANALYSIS TRANSFORMATION

The transformation for a nonblocking call is similar to the transformation for the `fork` call. However, a `nonblock` call can return a value, and the return value needs to be handled appropriately.

We transform a call

`x = nonblock foo(args)@`$c_1, \ldots, c_n$

to the program segment shown in Figure 17. In the figure, the list $v_1, v_2, \ldots, v_m$ is an over-approximation of the list of values that can be returned by `foo`. This can be computed by doing a flow-insensitive analysis over the body of `foo`.

C. COMPILER AND RUNTIME

Waiting Functions. We will use the following definitions: a *waiting function* is any function that contains a `waitfor` statement or (transitively) calls a function that contains a `waitor` statement; a *waiting call* is any call to a waiting function. We will treat `waitfor` as a waiting function that returns `void`.

At a waiting call site, it is possible for execution to be suspended and resume later in a different context. Therefore, at each waiting call site, it is necessary for the CLARITY runtime to collect enough information about the current context to resume execution: in particular, we need to preserve the values of function-local variables and the next statement to execute when the waiting call returns. These values are typically stored in an activation record on the call stack—since execution may resume in another physical thread, we cannot assume that the call stack will continue to be available unmodified.

Moving local variables to the heap. In event-driven programs, state is commonly managed using heap-allocated *control blocks*. These blocks are typically pre-allocated, fixed-size structures containing all of the information needed to resume execution at a later time. The programmer must design and manage these structures himself.

CLARITY provides the programmer similar functionality in an automatated fashion. The compiler transforms each waiting function to declare a `locals` structure containing its local variables. All references to local variables are transformed to refer to the `locals` structure, e.g., the assignment "`p = &x`" (where `p` and `x` are local variables) is transformed to "`locals->p = &(locals->x)`". Each waiting call is augmented with a `locals` parameter; the structures are chained

$$\textsc{Call-Fork}\ \frac{\langle M, E, Q, P \uplus \langle \texttt{fork f()}; S, K, L\rangle\rangle}{\langle M, E, Q, P \uplus \{\{\langle S, K, L\rangle, \langle \texttt{f()}, \bullet, \emptyset\rangle\}\}\rangle} \qquad \textsc{Call-Blk}\ \frac{\langle M, E, Q, P \uplus \langle \texttt{x = block f()}; S, K, L\rangle\rangle}{\langle M, E, Q, P \uplus \langle \texttt{f()}, (\texttt{b|k x}.S); K, L\rangle\rangle}$$

$$\textsc{Call-Nbl}\ \frac{\langle M, E, Q, P \uplus \langle \texttt{x = nonblock f()}; S, K, L\rangle\rangle}{\langle M, E, Q, P \uplus \langle \texttt{f()}, (\texttt{nb| x}.S); K, L\rangle\rangle} \qquad \textsc{Call}\ \frac{\langle M, E, Q, P \uplus \langle \texttt{f()}, K, L\rangle\rangle \qquad S \text{ is the body of } \texttt{f}}{\langle M, E, Q, P \uplus \langle S, K, L\rangle\rangle}$$

$$\textsc{Return-Blk}\ \frac{\langle M, E, Q, P \uplus \langle \texttt{return } v, (\texttt{b|k x}.S); K, L\rangle\rangle}{\langle M, E, Q, P \uplus \langle \texttt{x = } v; S, K, L\rangle\rangle} \qquad \textsc{Return-Nbl}\ \frac{\langle M, E, Q, P \uplus \langle \texttt{return } v, (\texttt{nb| x}.S); K, L\rangle\rangle}{\langle M, E, Q, P \uplus \langle \texttt{x = } v; S, K, L\rangle\rangle}$$

$$\textsc{Return-Empty}\ \frac{\langle M, E, Q, P \uplus \langle \texttt{return } v, \bullet, L\rangle\rangle}{\langle M, E, Q, P\rangle} \qquad \textsc{Send}\ \frac{\langle M, E, Q, P \uplus \langle \texttt{send } e; S, K, L\rangle\rangle}{\langle M, E \cup \{e\}, Q, P \uplus \langle S, K, L\rangle\rangle}$$

$$\textsc{SendAll}\ \frac{\langle M, E, Q, P \uplus \langle \texttt{sendall } e; S, K, L\rangle\rangle}{\langle M, E, Q', P' \uplus \langle S, K, L \cup \{e\}\rangle\rangle} \begin{array}{l}\text{where } \langle b_1, E_1, S_1, K_1, L_1 \cup \{e\}\rangle \in Q' \iff \langle b_1, E_1, S_1, K_1, L_1\rangle \in Q \\ \text{and } \langle S_2, K_2, L_2 \cup \{e\}\rangle \in P' \iff \langle S_2, K_2, L_2\rangle \in P\end{array}$$

$$\textsc{WaitFor-Sat}\ \frac{\langle M, E_1 \cup E_2, Q, P \uplus \langle \texttt{waitfor } r\ (W \cup (\ell, b, L_1 \cup E_1)); S, K, L_1 \cup L_2\rangle\rangle \qquad \overline{M}(b) \neq \textbf{false}}{\langle M, E_2, Q, P \uplus \langle \texttt{waitevent = } \ell; S, K, L_2\rangle\rangle}$$

$$\textsc{WaitFor-Blk}\ \frac{\begin{array}{c}\langle M, E_1, Q, P \uplus \langle \texttt{waitfor } r\ W; S, k_1; \ldots; k_n; \bullet, L\rangle\rangle \\ k_i = \texttt{b|k x}_i.S_i',\ 1 \leq i \leq n \qquad \forall \langle \ell, b, E\rangle \in W : \overline{M}(b) = \textbf{false} \vee E \not\subseteq E_1 \cup L\end{array}}{\langle M, E_1, Q \uplus \langle W, S, k_1; \ldots; k_k; \bullet, L \backslash E_2\rangle, P\rangle}$$

$$\textsc{WaitFor-Nbl}\ \frac{\begin{array}{c}\langle M, E_1, Q, P \uplus \langle \texttt{waitfor } r\ W; S_1, k_1; \ldots; k_n; (\texttt{nb| x}.S_2); K, L\rangle\rangle \\ k_i = \texttt{b|k x}_i.S_i',\ 1 \leq i \leq n \qquad \forall \langle \ell, b, E\rangle \in W : \overline{M}(b) = \textbf{false} \vee E \not\subseteq E_1 \cup L\end{array}}{\langle M, E_1, Q \uplus \langle W, S_1, k_1; \ldots; k_n; \bullet, \emptyset\rangle, P \uplus \langle \texttt{x = } r; S_2, K, L\rangle\rangle}$$

$$\textsc{Unblock}\ \frac{\langle M, E_1 \cup E_2, Q \uplus \langle W \cup \langle \ell, b, L_1 \cup E_1\rangle, S, K, L_1 \cup L_2\rangle, P\rangle \qquad \overline{M}(b) \neq \textbf{false}}{\langle M, E_2, Q, P \uplus \langle \texttt{waitevent = } \ell; S, K, L_2\rangle\rangle}$$

Figure 16: Semantic rules for CLARITY programs.

in a list, creating a shadow image of the call stack. When a thread blocks, a pointer to the current `locals` structure is saved on the wait queue. (Note that the `locals` structure does not contain the return address of the waiting call, because the address is not explicitly available at the C source code level.) This transformation is similar to the compilation strategy advocated by Appel and Zhao [4].

Continuation-passing transformation. The CLARITY compiler augments each waiting call with a continuation parameter, representing the next statement to be executed when control returns from the call. The continuation argument from the caller becomes part of the local environment for the waiting function. Since C does not directly support continuations, we modify the procedural structure of the source program by splitting a waiting function at each waiting call. E.g., a function `f` with a waiting call to `g`, "`f() { A; g(); B }`", will be translated into two functions `f0` and `f1` (we elide the `locals` structure argument, which is necessary to maintain the function-local state):

```
f0() { A; g(f1); }
f1() { B }
```

In addition, any `return` statement in a waiting function must be transformed to instead invoke a continuation on the caller's `locals` structure. This transformation is applied recursively to each waiting call in a function. The precise details of the transformation, including the handling of local and inter-procedural control flow, are standard (see, e.g., Appel [3]) and beyond the scope of this paper.

Taken together, the local variable and continuation-passing transformations automate the translation from a threaded execution model to event-driven code—they allow a blocked thread to be resumed at any time, from any calling context, by simply invoking the thread's continuation on the thread's `locals` structure.