Encapsulating Concurrency as an Approach to Unification

Santosh Kumar[*], Bruce W. Weide[*], Paolo A.G. Sivilotti[*], Nigamanth Sridhar[†], Jason O. Hallstrom[‡], Scott M. Pike[#]

[*] The Ohio State University, Computer Science & Engineering, {kumars,weide,paolo}@cse.ohio-state.edu
[†] Cleveland State University, Electrical & Computer Engineering, n.sridhar1@csuohio.edu
[‡] Clemson University, Computer Science, jasonoh@cs.clemson.edu
[#] Texas A&M University, Computer Science, pike@cs.tamu.edu

ABSTRACT

We extend traditional techniques for sequential specification and verification to systems involving intrinsically concurrent activities. Our approach uses careful design of component specifications to encapsulate inherent concurrency, and hence isolate clients from associated verification concerns. The approach has three parts: (i) relational specifications to capture the interleaved effects of concurrent threads of execution, (ii) intermediate components to support a client's view of being the only active thread of computation, and (iii) a new specification clause to express requirements on a client's future behavior. We illustrate these ideas, and discuss their merits, in the context of a case study specified using RESOLVE.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification—*Methodologies*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Design By Contract*

General Terms

Verification

Keywords

Unification problem, sequential verification techniques, concurrent systems, relational specification, mutual exclusion, expects clause.

1. INTRODUCTION

Design-by-contract [10] has long been recognized as a foundation for component-based specification and verification. For sequential systems, contract specification mediates interactions between a component and its *client*. By contrast, for concurrent systems the contract specification is between a component and its *environment*. The difference is significant since in a sequential system, the client and component operate in the same thread of execution, whereas in a concurrent system, the environment may include other threads of execution. Taking a sequential system view leads to traditional input-output specifications and Hoare-style verification techniques. On the other hand, taking a concurrent system view leads to explicit progress requirements and guarantees, where circularities in reasoning must be carefully avoided. As a result, research in these nominally related areas has focused on separate and largely orthogonal issues. There has been remarkably little cross-fertilization.

Specification and verification techniques for sequential systems (particularly component-based systems) generally involve showing that, given acceptable input values, a computation produces specified output values. The operational model is normally a standard von Neumann machine. Some issues that arise include (1) the contract style (*e.g.*, algebraic or model-based), (2) the impact of aliased references on sound reasoning, and (3) the difficulties introduced by modern programming language constructs (*e.g.*, inheritance, user-defined types, etc).

Specification and verification techniques for concurrent systems generally involve showing that cooperating processes adhere to specified temporal properties. Models of computation may vary, with message passing or shared memory, various degrees of synchrony, etc. Programming language features and variable types tend to be simpler in order to abstract away from complex details concerning "what is being computed" in the more traditional sense. Still, there are new communication constructs that are not present in the sequential case. Some issues that arise include (1) the expressiveness of various temporal logics, (2) the non-determinism introduced by interleaved access to shared resources, (3) progress and fairness properties for scheduling individual processes, and (4) non-interference requirements to guarantee sound proof systems, etc.

1.1 The Unification Problem

The *unification problem* in the title refers to the following impediment to fully successful specification and verification of component-based software: There is a need for a unified theory that reconciles work in the now largely disjoint research areas of specification and verification of sequential systems, and of concurrent systems. This problem is important because modern software systems involve both aspects. A typical modern program is charged not only with com-

Specification and Verification of Component-Based Systems (SAVCBS'04) Workshop at the 12^{th} ACM SIGSOFT Symposium on Foundations of Software Engineering. Newport Beach CA, USA. Oct 31 – Nov 1, 2004.

puting specified data values, but with doing so reliably in the context of concurrent threads of activity that may be accessing resources that it shares with them.

This paper partially develops an approach to addressing the unification problem. Based on an early case study, we believe it will permit extending the realm of traditional sequential specification and verification techniques to systems that involve intrinsically concurrent activities. The idea is to use certain component design and contract specification techniques to create what appear to be ordinary sequential components for the purposes of client interaction, and to "bury" or encapsulate concurrency inside the implementations of those components. This shields otherwise sequential clients of shared resources from the complications normally associated with concurrency. Among other things, for example, specification and verification of client correctness need not be extended from the sequential situation by introducing temporal logic whenever the client also has to deal with concurrent activities.

To give the flavor of the approach—and to emphasize that we are claiming novelty from the standpoint of specification and verification issues and *not* from the standpoint of, say, a new software design pattern—consider a nominally sequential program that needs to use a printer. The printer is a shared physical device that other programs like it might also be using. Without some method to encapsulate a protocol for ensuring mutually exclusive access to the printer, each program that shares it must understand how to negotiate access with the printer's other clients. This necessarily involves modeling the other clients, specifying liveness and safety properties associated with mutual exclusion, detailing some particular protocol for mutual exclusion, etc. But if there is a carefully-designed printer-monitor component that allows each client to create its own logical printer object and simply print to it as though it were alone in the world, all the complications associated with sharing—indeed, all the problems associated with specifying and reasoning about the concurrency involved in accessing the physical printer—are moved down one level in the system. The clients of the printer-monitor component see an ordinary sequential component. The problem of concurrency is not moved over the horizon, just down one level. But the impact is that there are more clients for which sequential contract specifications and verification techniques can be used.

This is, of course, just one possible approach to the unification problem. It is premature to discuss whether it might be the best one or whether it could ultimately lead to a complete solution to the problem. Instead, the goal of this paper is to report on progress and to describe some open issues in the hope that others will step forward to address the unification problem, too.

Paper Organization. The rest of the paper is organized as follows. Section 2 outlines the main technical features of our approach. Section 3 presents the case study. Section 4 sketches some related work. Section 5 concludes the paper and Section 6 points to open issues.

2. TECHNICAL FEATURES

There are three main technical features to the approach we present. Individually, each is not new; the primary contribution is combining them in a special new way. Each feature is highlighted during the case study in Section 3.

2.1 Using Relational Specifications

The first idea is to use relational specifications to model potential interference from concurrent threads of execution. In this way, each client of a component can maintain an especially simple view of the system: it is as though there are no other threads of execution. The hoped-for result is that specification and reasoning can be done using ordinary techniques for sequential systems — so long as they admit relational behavior (*i.e.*, are not restricted to purely functional input-output specifications).

There is an interesting connection between relational specifications and the philosophical theory of *epistemic solipsism*. Roughly stated, this doctrine draws a methodological distinction between ontology (that which exists) and epistemology (that which we can know exists). Epistemic solipsism witnesses a gap between these concepts, such that it is possible for certain entities to exist, despite our inability to know them. We make no judgment here about the coherence of solipsism as a philosophical doctrine, but only about the similarity in principle between solipsism and the use of relational specifications to mask concurrency.

By encapsulating concurrency inside an observably sequential component, we seek to create a cover story that makes it *appear to the client programmer* as though all state changes in all objects are the result of method calls made by the client. In particular, no object's value ever changes spontaneously; indeed, nothing changes unless the client initiates such a change. Concurrency may *exist* in the implementations of such components, but it cannot be *known* to the client insofar as it is unobservable through any component interface. Consequently, there is never a need for the client programmer to postulate the existence of concurrent threads of activity to explain or verify the correctness of the observed behavior with respect to the given sequential specifications.

2.2 Separating Proxy and Core Components

Direct application of the above idea in principle might lead to a sequential-component contract specification for the client. But it is not necessarily a pretty one. One can make visible, in the client's nominally sequential view of behavior, all the state needed to capture the underlying concurrency in the implementation of the component. The specification merely says that this state changes in bizarre though technically explainable ways. But probably most of this state is not needed to explain the same behavior, *i.e.*, the specification is not fully abstract. So, the second idea is to simplify the contract specification for the client by introducing an intermediate component that acts as a proxy in client dealings with the underlying component that encapsulates concurrency (which we call the core component).

Figure 1 illustrates how this works in the case study. Round-cornered boxes stand for contract specifications, rectangles for implementations, and arrows for the relationships on their labels. For example, the top box represents the client specification, which involves only sequential constructs. The rectangle below it stands for the implementation of this specification—what we have been calling the client program. This program claims to implement the specification above it, which is something that needs to be verified. It uses some implementation of the Mutex_Proxy specification, which in turn hides the concurrency inherent in the Mutex_Core specification below it. The client program could, in principle, directly use something like the specification Mutex_Core (although the specification details would be quite different than those developed in the case study). However, that specification would be unnecessarily complex compared to the much simpler Mutex_Proxy specification.

2.3 Additional Client Obligations

In reasoning about the correctness of implementations of the core and proxy components, one is faced with the usual verification situation for traditional sequential components: clients of a contract specification are responsible for establishing preconditions on calls, implementations are responsible for establishing postconditions. However, because of the need to reason about properties that otherwise would involve temporal logic specifications, we must introduce an additional reciprocal obligation on every client of the proxy. This is needed to show that *other* clients will make progress. The objective of proving total correctness of the client cannot be achieved if other clients—whose very existence is not knowable to any one of them—can thwart termination of any of the calls to proxy methods.

The third idea, then, is to introduce an additional specification construct in the sequential specification language. We call this new construct the expects clause. While a method's requires clause defines an obligation the client must meet *before* calling the method, its expects clause describes an obligation the client must meet at some point *after* calling the method. This obligation is given as a set of method calls that must be made in the future.

For example, consider the following path expression specification for a read-only file:

```
(Open; Read*; Close)
```

This expression stipulates that once a file has been opened, the Read() operation can be invoked several times (or never at all), and finally the file must be closed. That is, as a consequence of invoking Open(), the client is required to invoke Close() in the future. This obligation could be reflected in the contract as follows:

```
operation Open()
    requires true
    ensures self.Is_Open
    expects self.Close()
```

A brief operational sketch of the semantics of the expects clause is as follows. In addition to the program context and variable values in each state of a program, we maintain a "promises set" PS of all the method calls that the program has promised to other components via expects clauses, and has not yet discharged. Whenever an operation is invoked, it is removed from PS, if it is there. If the call is to an operation that has its own expects clause, the calls in that clause are added to PS. For other statements, PS is not modified. At termination, if the program has honored all the promises it made, then PS is empty. Therefore, the client, in addition to dispatching proof obligations for all pre-conditions, also has to dispatch an additional proof obligation that PS is an empty set at the point of termination. (Please see Section 6 for issues raised by this over-simplified view.)

3. CASE STUDY: MUTUAL EXCLUSION

In this section, we present our specification of a mutual exclusion component we call Mutex, using the RESOLVE

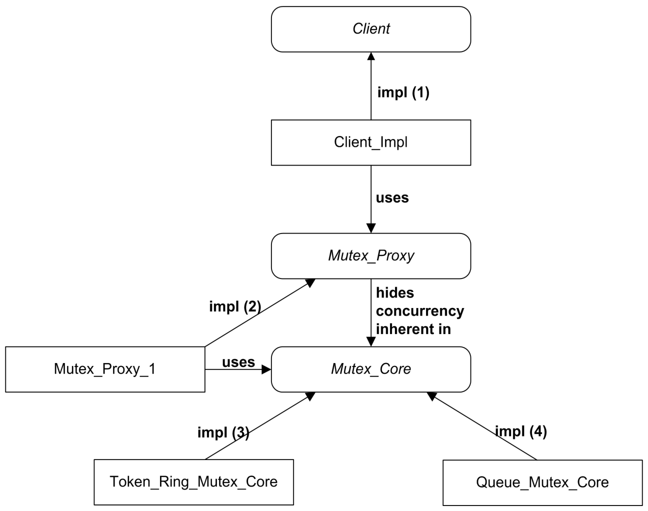

Figure 1: Component coupling diagram (CCD) for a system using the Mutex component

specification language [7]. This single-object component is divided into two parts — the first, called Mutex_Proxy, presents a sequential view to the client, and the second, called Mutex_Core, encapsulates a conflict resolution protocol for mutual exclusion. All aspects of concurrency in the conflict resolution protocol are encapsulated inside Mutex_Core, and do not bleed through to the client. In effect, there are several instances of Mutex_Proxy in a distributed system (one for each client), all of which interact with a single instance of Mutex_Core. The design of this system structure is shown in Figure 1.

3.1 Mutex_Proxy

Listing 1 shows the specification of our Mutex_Proxy abstract component. The mathematical model of the Mutex_Proxy type is defined by MUTEX_PROXY_MODEL (Lines 8—14). This is a 3-tuple of two booleans, requested and available, and a NATURAL_NUMBER, wait_index. If the client that uses this proxy has requested access to the critical section, requested becomes **true**, and when it is safe for the client to access the critical section, available becomes **true**. The purpose of wait_index is to allow a client to prove its progress, *i.e.*, if it requests access to the resource, it will eventually have it available. For a given proxy, available can only become **true** if requested is also **true** (the client has requested access to the critical section). Upon initialization, every instance of Mutex_Proxy has both requested and available equal to **false** and wait_index equal to 0 (Lines 16—18).

The Mutex_Proxy component exports the following operations:

Request() **(Lines 20—26):** A client invokes the Request() method when it wants access to the critical section. The various preconditions make sure that this method can only be invoked once per "cycle" — once a client requests access, it cannot make another request until it has either used the critical section and then released it, or simply canceled its request by calling Release(). The post-condition of the Request() operation says that requested becomes **true** and wait_index assumes some natural number value (which, it turns out, the client has no way to observe, except to know whether it is zero). In addition, as a consequence of this invocation, the client is committed to invoking Release() in every possible future computation.

Is_Requested() **(Lines 28—30):** This operation can be invoked to check whether the client has requested access to the resource. (It is provided for functional completeness but plays no other substantive role.)

Check_If_Available() **(Lines 32—40):** After a client has requested access to the resource, a call to this procedure returns with b equal to **true** when the client can safely access the resource. Also, with every invocation of this method, the value of wait_index decreases if it is positive. Finally, if and only if wait_index is already 0, then it stays the same and available becomes **true** (as does b). Notice that Check_If_Available(), by virtue of its relational specification, has the property that (sometimes) it makes the client believe that available changes *as a result* of the Check_If_Available() call. That is, there is no reason for the client to explain the change in the value of available by postulating the existence of some other process having concurrent access; the client's call to Check_If_Available() is what has *caused* available to change.

Release() **(Lines 42—46):** The Release() operation is invoked to signal the end of the critical section or to cancel an outstanding request. This operation results in requested and available both becoming **false**.

3.2 Mutex_Core

Listing 2 shows the mathematical model of Mutex_Core.

PROXY_SET **(Lines 3—4):** This denotes a mathematical set of proxy identities, each of which is a natural number.

WAITING_PROXY **(Lines 6—8):** This denotes a pair of natural numbers, id to represent the identity of the proxy, and ticket to represent some metric that determines when the proxy will get access to the resource.

WAITING_PROXY_SET **(Lines 10—16):** This denotes a set of waiting proxies, *i.e.*, those that have requested access to the resource.

MUTEX_CORE_MODEL **(Lines 18—35):** The mathematical model of the Mutex_Core type is a tuple that consists of the set, all_proxies, of all the proxies that have "registered" with this Mutex_Core instance; the set, waiting_proxies, of just those proxies in all_proxies that have requested access to the resource but have not yet released it or canceled the request; and the integer, current_id, which is the (non-negative integer) identity of the proxy that currently has access to the resource because it has a minimum ticket value among all waiting proxies (or a negative number if there are no waiting proxies). Notice that a proxy can request access to a resource only after registering with the Mutex_Core instance that is responsible for that resource. Upon initialization, a Mutex_Core instance has no proxies registered (and hence no waiting proxies) and current_id is -1 (no proxy is accessing the resource).

Listings 3 and 4 present the specifications of the operations that the Mutex_Core component exports.

Listing 1: Specification of abstract Mutex_Proxy component

```
1  abstract component Mutex_Proxy
2  
3  math subtype NATURAL_NUMBER is integer
4    exemplar n
5    constraint
6      n >= 0
7  
8  math subtype MUTEX_PROXY_MODEL is
9    (requested: boolean,
10    available: boolean,
11    wait_index: NATURAL_NUMBER)
12   exemplar mpm
13   constraint
14     if mpm.available then mpm.requested
15 
16 Mutex_Proxy is modeled by MUTEX_PROXY_MODEL
17   initialization ensures
18     self = (false, false, 0)
19 
20 procedure Request ()
21   requires
22     not self.requested
23   ensures
24     there exists i: NATURAL_NUMBER such that
25       (self = (true, false, i))
26   expects self.Release()
27 
28 function Is_Requested (): Boolean
29   ensures
30     Is_Requested = self.requested
31 
32 procedure Check_If_Available (replaces b: Boolean)
33   requires
34     self.requested
35   ensures
36     self.requested and
37     (if #self.wait_index /= 0
38      then self.wait_index < #self.wait_index
39      else #self.wait_index = self.wait_index) and
40     b = self.available = (self.wait_index = 0)
41 
42 procedure Release ()
43   requires
44     self.requested
45   ensures
46     self = (false, false, 0)
```

Listing 2: Specification of abstract Mutex_Core component

```
abstract component Mutex_Core

math subtype PROXY_SET is
    finite set of NATURAL_NUMBER

math subtype WAITING_PROXY is
  (id: NATURAL_NUMBER,
   ticket: NATURAL_NUMBER)

math subtype WAITING_PROXY_SET is
    finite set of WAITING_PROXY
  exemplar wps
  constraint
    for all p,q: WAITING_PROXY
        where ({p,q} is subset of wps)
      (if p.id = q.id then p = q)

math subtype MUTEX_CORE_MODEL is
  (all_proxies: PROXY_SET,
   waiting_proxies: WAITING_PROXY_SET,
   current_id: integer)
  exemplar mcm
  constraint
    for all p: WAITING_PROXY
        where (p is in mcm.waiting_proxies)
      (p.id is in mcm.all_proxies) and
    if mcm.waiting_proxies = {}
      then mcm.current_id < 0
      else
        there exists p: WAITING_PROXY such that
          (p is in mcm.waiting_proxies and
           mcm.current_id = p.id and
           (for all q: WAITING_PROXY
                where (q is in mcm.waiting_proxies)
              (q.ticket >= p.ticket)))

Mutex_Core is modeled by MUTEX_CORE_MODEL
  initialization ensures
    self = ({}, {}, -1)
```

Listing 3: Operations of abstract Mutex_Core component

```
math definition IS_REQUESTED
    (id: integer, wps: WAITING_PROXY_SET): boolean
  explicit definition
    there exists wp: WAITING_PROXY such that
      ((wp is in wps) and (wp.id = id))

math definition MIN_TICKET
    (wps: WAITING_PROXY_SET): NATURAL_NUMBER
  explicit definition
    min ({0} union {wp: WAITING_PROXY
         where (wp is in wps) (wp.ticket)})

procedure Add_Proxy (replaces id: Integer)
  ensures
    self.waiting_proxies = #self.waiting_proxies and
    self.current_id = #self.current_id and
    id is not in #self.all_proxies and
    self.all_proxies = #self.all_proxies union {id}
```

Listing 4: Operations of abstract Mutex_Core component (contd.)

```
procedure Remove_Proxy (evaluates id: Integer)
  requires
    id is in self.all_proxies and
    not IS_REQUESTED(id, self.waiting_proxies)
  ensures
    self.waiting_proxies = #self.waiting_proxies and
    self.current_id = #self.current_id and
    self.all_proxies = #self.all_proxies - {id}

procedure Request (evaluates id: Integer)
  requires
    id is in self.all_proxies and
    not IS_REQUESTED(id, self.waiting_proxies)
  ensures
    self.all_proxies = #self.all_proxies and
    there exists ticket: NATURAL_NUMBER such that
      ((if #self.waiting_proxies = {}
        then (ticket = 0 and self.current_id = id)
        else (ticket >
                MIN_TICKET(self.waiting_proxies) and
              self.current_id =
                    #self.current_id)) and
       self.waiting_proxies =
         #self.waiting_proxies union
           {(id, ticket)})
  expects self.Release(id)

function Is_Requested (id: Integer): Boolean
  requires
    id is in self.all_proxies
  ensures
    Is_Requested =
      IS_REQUESTED(id, self.waiting_proxies)

procedure Check_If_Available
    (evaluates id: Integer, replaces b: Boolean)
  requires
    id is in self.all_proxies and
    IS_REQUESTED(id, self.waiting_proxies)
  ensures
    self.all_proxies = #self.all_proxies and
    self.current_id = #self.current_id and
    b = there exists wp: WAITING_PROXY such that
        (wp is in #self.waiting_proxies and
           id = wp.id = self.current_id)

procedure Release (preserves id : integer)
  requires
    id is in self.all_proxies and
    IS_REQUESTED(id, self.waiting_proxies)
  ensures
    self.all_proxies = #self.all_proxies and
    there exists wp: WAITING_PROXY such that
      (wp is in #self.waiting_proxies and
       wp.id = id and
       self.waiting_proxies =
         #self.waiting_proxies - {wp}) and
    if self.waiting_proxies = {}
    then self.current_id = -1
    else
      (self.current_id,
       MIN_TICKET (self.waiting_proxies)) is in
         self.waiting_proxies
```

Useful Mathematical Definitions

IS_REQUESTED **(Lines 1—5):** Is the proxy with the given id in the set of waiting proxies?

MIN_TICKET **(Lines 7—11):** The minimum value of ticket in the set of waiting proxies.

Operations

Add_Proxy() **(Lines 13—18):** This operation adds a new proxy to the set of proxies that want to use the resource. The operation ensures the uniqueness of id's in the set of all proxies. It returns the id of the newly created proxy.

Remove_Proxy() **(Lines 1—8):** Given an id, this operation removes the proxy with this id. The proxy can no longer request access to the resource once it is removed.

Request() **(Lines 10—26):** When this operation is invoked by a valid proxy, the proxy is given a ticket, whose value can not be directly observed by any proxy. The proxy's id and the associated ticket are added to the set of waiting proxies. If, at the time of invocation, no other proxy had requested access to the resource, the proxy invoking Request() gets access to the resource, *i.e.*, the current_id becomes this proxy's id. This means that the proxy will be able to access the resource immediately after its first call to Check_If_Available(). If there are pending requests from other proxies at the time of invocation, the ticket assigned is some larger value than MIN_TICKET(waiting_proxies).

Is_Requested() **(Lines 28—33):** This operation returns **true** if the given id belongs to a waiting proxy, and **false** otherwise.

Check_If_Available() **(Lines 35—45):** After requesting access to the resource, a proxy invokes this operation to see if the resource is available to it. The procedure returns with b equal to **true** if the given id is equal to current_id and the waiting proxy with this id has a ticket value equal to MIN_TICKET(waiting_proxies).

Release() **(Lines 47—63):** Once a proxy is done using the resource, it invokes Release(). This removes it from the set of waiting proxies, and results in current_id becoming either -1 (in case there are no other waiting proxies), or the id of some still-waiting proxy whose ticket is equal to MIN_TICKET(waiting_proxies). The next time the proxy whose id equals current_id invokes Check_If_Available(), it will be granted access to the resource.

3.3 Proofs of Implementations

As shown in Figure 1, there are four *implements* relations. Each of these has a set of associated proof obligations to show that the implementations meet the specifications. We focus on the new aspects of these (the *impl(1)* and *impl(2)* relations in Figure 1) in this paper and defer the other two, which will be similar to each other, for future work.

3.3.1 Proof of Client Progress

We notice that the proof of the client implementation will be similar to any sequential component, although it is using a concurrent program. As mentioned in Section 2.1 and Section 2.2, the verification for client implementation is reasonably simple because it is using the sequential specification of the proxy as opposed to that of the core. For example, the client will be able to prove loop termination in the following code snippet:

```
proxy1.Request();
while(not b) {
    proxy1.Check_If_Available(b); }
proxy1.Release();
```

When it calls Request() on proxy1, which is an instance of Mutex_Proxy_1, wait_index assumes the value of some natural number. With every call to Check_If_Available() this number goes down. Eventually, it will hit zero and the client will be able to access the critical section. We also notice that the client is able to prove, based on the termination of the loop, that it satisfies the expects obligation it acquired by calling Request().

3.3.2 Proof of Proxy Implementation

The key part to prove in this proof obligation is that the proxy can meet the ensures clause of Check_If_Available() by using the specification of Mutex_Core.

A sketch of the implementation of the Mutex_Proxy is as follows: In the constructor, it calls Add_Proxy() and gets an *id*. In Request(), it calls Request(id) on Mutex_Core and sets its requested to true. In the Check_If_Available(), it sets b and available to true, if it gets a true answer from the Mutex_Core. In Release(), it calls the corresponding operation of the Mutex_Core and sets both requested and available to false.

In the following lemmas and theorems, we assume that mc is the common instance of a Mutex_Core implementation that all proxies are using. Further, we use MP to denote the Mutex_Proxy type. We also use min_ticket to denote MINTICKET(mc.waiting_proxies). Lastly, we use the terms "getting access to the resource" for a proxy synonymously with the event that the associated *id* becomes equal to mc.current_id.

LEMMA 3.1. $\forall$ wp$\in$ mc.waiting_proxies, (wp.ticket-min_ticket,k)*, where* k *is the number of proxies with ticket equal to* min_ticket*, decreases with every call to* mc.Release() *by proxies who get access to the critical section, until* mc.current_id = wp.id.

PROOF. The value of wp.ticket is unchanged once it is assigned by mc.Request(). If |mc.waiting_proxies| = 0 at the time wp calls mc.Request(), it follows from the postcondition of Request() that mc.current_id = wp.id. On the other hand, if at the time of wp's call to mc.Request(), |mc.waiting_proxies| > 0, then (from the constraint in lines 29-35 in Listing 2), as long as |mc.waiting_proxies-wp|> 0, there is some proxy ap such that mc.current_id = ap.id. From the expects clause of mc.Request(), ap eventually calls mc.Release(). With this call to mc.Release(), either min_ticket increases (if there were no other waiting proxies with the same value of ticket as ap.ticket) or k decreases. In either case, the tuple (wp.ticket-min_ticket,k) decreases[1]. The same situation now

[1]We use the standard ordering relation on tuples, where (a, b) $<$ (c, d) iff (a $<$ c) or a = c and b $<$ d.

recurs with some other proxy np taking the place of ap until mc.current_id = wp.id. This completes the proof. □

THEOREM 3.1. *∀ wp ∈ mc.waiting_proxies eventually mc.current_id = p.id, unless wp cancels its request by calling mc.Release() prematurely.*

PROOF. From Lemma 3.1, (wp.ticket - min_ticket, k) decreases with every call to mc.Release() by proxies who get access to the resource before wp. When the value of (wp.ticket - min_token, k) becomes (0, 0), wp is guaranteed to have mc.current_id = p.id, although it may get access to the resource earlier when min_ticket = 0 and k > 0. □

Theorem 3.1 ensures that every proxy can meet the ensures clause of its Check_If_Available() method. A more formal proof of this claim can be done by using an abstraction relation [12] to relate wait_index to its associated ticket in mc.

3.4 Proof of Safety and Starvation Freedom

The entire mutual exclusion program is making some guarantees to the system designer. These are the safety and progress guarantees [2]. The safety specification says that "at most one client gets access to the critical section at a time." The progress specification says that "every requesting client eventually gets access to the critical section." The system uses the Mutex_Proxy and Mutex_Core components to meet these specifications.

The progress property follows immediately from Theorem 3.1. We prove the safety property below. In the following, we assume that the *id* assigned by the Mutex_Core to a proxy is one of the fields of proxy. We use the definitions and assumptions from Section *3.3.2*.

LEMMA 3.2. *The following invariant holds in the system: ∀ p ∈ MP : p.available⇒ (p.id = mc.current_id).*

PROOF. The invariant vacuously holds at initialization since available is false for all proxies. Now, when p.available changes from false to true (in a call to mc.Check_If_Available()) for some proxy p, it is only when p.id = mc.current_id. Therefore, the invariant is preserved by this transition. When the assertion p.id = mc.current_id changes from true to false (in a call to mc.Release()), p.available is set to false, thus preserving the invariant in this transition too. Therefore, the invariant holds. □

THEOREM 3.2. *∀ p ∈ MP : p.available ⇒ (∀ op ∈ MP : op ≠ p ⇒ ¬ op.available).*

PROOF. The theorem holds vacuously if ∀ p ∈ MP : ¬ p.available. So, let us assume ∃ p ∈ MP: p.available. From Lemma 3.2, p.id = mc.current_id. Further, from the constraint on Line 16, ids are unique. Now, the assignment p.id = mc.current_id is done either in mc.Request() or in mc.Release(). If the assignment was done in mc.Request(), then there were no other waiting proxies and p is the unique proxy with available set to true. If the assignment was done in mc.Release(), then the proxy with available set to true previously, had set its available to false at the time that mc.current_id was assigned to p.id. Therefore, in this case also p is the unique proxy with its available set to true. From the uniqueness of ids, no other proxy can get an affirmative answer in a call to mc.Check_If_Available(), as long as p does not call mc.Release(). Therefore, p remains the unique proxy with its available set to true as long as it does not call mc.Release(). When p calls mc.Release(), either there are no more waiting proxies, in which case no proxy has its available set to true, or some waiting proxy q takes the place of p. This completes the proof. □

4. RELATED WORK

Concurrent systems often exhibit reactive behavior, so specification techniques for such systems usually include explicit treatment of both safety and progress properties. Many different compositional approaches have been investigated, including rely-guarantee [13, 1, 8], hypothesis-conclusion [4], and assumption-commitment [5]. All of these techniques address the problem of circularities in proofs in some way, for example by restricting the properties on which a component relies to be safety properties only. In contrast, the expects clause introduced here allows a component to explicitly require a progress property of its client. Circularities are avoided by preventing a client from requiring any progress properties from the components it uses, apart from termination of each called method.

A common way to model concurrency is with nondeterministic interleaving, as in Unity [4] and TLA [9]. Our use of relational specifications to capture the potential effects of concurrently executing threads of execution is certainly not new. The introduction of an intermediary component (the proxy), however, facilitates a solipsistic view on the part of a client, and in this way promotes a novel way to compositionally reason about program behavior.

The Seuss methodology [11] is similar to our proposed approach in that it draws on both concurrent and sequential techniques. Whereas we begin with a sequential framework (RESOLVE) and add elements to address concurrency, Seuss begins with a concurrent framework (similar to Unity) and incorporates program structures such as processes and methods with sequential invocation semantics.

5. CONCLUSION

We proposed an approach to unify the specification and verification of sequential systems with those of concurrent ones. We proposed a specification approach that characterizes concurrent programs as a pair of components — a Proxy and a Core component. The Proxy component uses relational specifications to present a sequential veneer over the Core component, allowing clients of a concurrent program to be verified without concern for concurrency. We also introduced a new expects clause in the contractual specification of operations to formalize the well-behavedness requirements from the clients. We illustrated our approach in the context of the traditional mutual exclusion problem. In our case study, we also demonstrated how the proof obligation associated with the expects clause can be carried out by the clients.

6. OPEN ISSUES

As the work presented in this paper is preliminary, there are a number of avenues for future investigation. We plan to investigate the utility of the expects construct in specifying other RESOLVE components. We also plan to investigate other possible versions of the mathematical structure involved in the operational semantics of the expects clause, i.e. multi-set, string or some other model in place of a set. Further, we plan to investigate what proof obligations a non-terminating program should have with respect to the expects clauses in the components it uses. We would like to point out here that it is tempting to make the structure and semantics of the expects clause very rich, but it is not yet clear whether that will be useful. For example, in our case study of the Mutex component, nested calls to methods with expects clause do not arise, although this may have been the case had we designed our component in a different way.

Another area of investigation is the evaluation of the applicability limits of our approach. A successful application of our approach to the case study of mutual exclusion program in this paper is evidence that our approach can work for conflict resolution programs, also called *competitive systems*, such as dining philosophers [6] and drinking philosophers [3]. Currently, we are in the process of applying this approach to some systems in the other domain of concurrent programming, *cooperative systems.* In particular, we are working on designing components for barrier synchronization and network protocol stack, both of which are representative of cooperative concurrent systems.

In the more distant future, we plan to focus on a proof system for verifying the correctness of concurrent component implementations.

7. ACKNOWLEDGMENTS

We thank the anonymous referees for their helpful insights, and especially for more suggestions for future work than we could elaborate here. We also thank the Reusable Software Research Group (RSRG) at the Ohio State University for their comments and criticisms on this work.

8. REFERENCES

[1] Abadi, M., and Lamport, L. Composing specifications. *TOPLAS 15*, 1 (Jan 1993), 73–132.

[2] Alpern, B., and Schneider, F. B. Defining liveness. *Information Processing Letters 21*, 4 (Oct 1985), 181–185.

[3] Chandy, K. M., and Misra, J. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst. 6*, 4 (1984), 632–646.

[4] Chandy, K. M., and Misra, J. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, USA, 1988.

[5] Collette, P. Composition of assumption-commitment specifications in a UNITY style. *SCP 23* (Dec 1994), 107–125.

[6] Dijkstra, E. W. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 2 (Oct 1971), 115–138.

[7] Edwards, S. H., Heym, W. D., Long, T. J., Sitaraman, M., and Weide, B. W. Specifying Components in RESOLVE. *Software Engineering Notes 19*, 4 (1994), 29–39.

[8] Jones, C. B. Tentative steps toward a development method for interfering programs. *TOPLAS 5*, 4 (1983), 596–619.

[9] Lamport, L. The temporal logic of actions. *TOPLAS 16*, 3 (May 1994), 872–923.

[10] Meyer, B. *Design by contract.* Prentice Hall, 1992, ch. 1.

[11] Misra, J. *A Discipline of Multiprogramming.* Monographs in Computer Science. Springer-Verlag, 2001.

[12] Sitaraman, M., Weide, B. W., and Ogden, W. F. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering 23*, 3 (1997), 157–170.

[13] Stark, E. W. A proof technique for rely guarantee properties. In *Foundations of software technology and theoretical computer science*, no. 306 in LNCS. Pergamon-Elsevier Science Ltd., 1985, pp. 369–391.