Distributed Diners: From UNITY Specification to CC++ Implementation

Ulla Binau*
Department of Computer Science
California Institute of Technology
ulla@cs.caltech.edu

Caltech-CS-TR-93-20
June 3, 1993

Abstract

Resource conflicts may typically be described as the dining philosophers problem (or diners for short). In this report we derive a distributed message-passing solution to the diners problem from the shared memory solution presented in [CM88, Ch.12 Dining Philosophers].

We define an isomorphism between variables in the shared memory state and variables in the distributed state. This allows us to translate the shared memory UNITY specification to a distributed UNITY specification without affecting the validity of the original refinement proof.

It turns out that the translated progress properties cannot be fulfilled by the solution scheme we have in mind. However, we show that weaker properties may be used instead, still without affecting the correctness of the original proof.

The derivation of a UNITY program from the translated properties is not quite obvious. Hence we introduce an extra refinement step prior to deriving our distributed UNITY implementation. Finally the distributed UNITY implementation is translated to Compositional C++, (CC++) a parallel extension of C and C++.

Note: The reader is assumed to be familiar with UNITY [CM88] and C, C++ or CC++ [KR88, Str91, CK92].

*visiting from Department of Computer Science at The Technical University of Denmark

Contents

1 Motivation and Background

The aim of this report is twofold:

- Illustrate stepwise refinement from a shared memory UNITY specification to a distributed CC++ implementation.
- Derive an executable distributed solution to the conflict resolution problems that can be described as the dining philosophers problem.

A resource conflict occurs when processes share resources such that each resource is shared among a fixed subset of the processes and each resource may be used by at most one process at any time. This type of problem is often referred to as the dining philosophers (or diners) problem. The problem has been formalized and solved for shared memory systems in e.g., [CM88, Ch.12]. We could start from scratch; formalize the problem and derive a solution. However, since this would imply re-doing much of the work done by Chandy and Misra, we find it more appealing to extend their refinement. Also, this would illustrate how to reuse existing solutions and their correctness proofs.

In the remainder of this section we will give an introduction to Chandy and Misra's work on the diners problem and hint at how we intend to extend their work to distributed systems.

1.1 The Diners Problem

1.1.1 Informal Description

Given a static, finite, undirected network or graph of processes, such that processes that share a resource are neighbors. Each vertex of the graph represents a philosopher (process). A philosopher can be thinking (doing work that does not require the shared resource), hungry (ready to use the shared resource), or eating (using the shared resource). Neighboring philosophers may not eat simultaneously. All philosophers are thinking initially. Each philosopher decides when he gets hungry and when he stops eating. He starts thinking, when he stops eating, and he can only get hungry after thinking.

Now the aim is to implement a control system, which guarantees that no neighbors eat simultaneously and that whenever a philosopher gets hungry, sooner or later he will start eating (under the assumption that no philosopher eats forever).

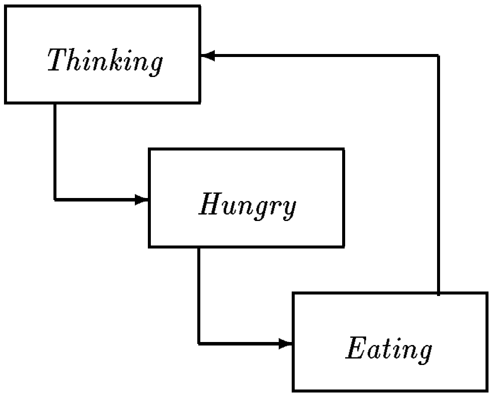

Figure 1: The philosopher state changes.

1.1.2 Formal Specification

Each philosopher is assigned a unique number u, such that all philosopher numbers form a partial order. The network is given as a boolean matrix E, where $E[u, v]$ means that there is an edge between u and v, i.e. they are neighbors.

The control system and philosopher u interacts via a shared variable $u.dine$, which reflects the state of the philosopher, and we define:

$$u.t \equiv (u.dine = t) \tag{1}$$
$$u.h \equiv (u.dine = h) \tag{2}$$
$$u.e \equiv (u.dine = e) \tag{3}$$

The behavior of the philosophers (u*ser of* d*inners*) is specified in UNITY [CM88, Sec.12.2] as follows:

$u.t$ **unless** $u.h$	**in** *user*	(udn1)
stable $u.h$	**in** *user*	(udn2)
$u.e$ **unless** $u.t$	**in** *user*	(udn3)
Conditional Property	**in** *user*	(udn4)

Hypothesis: **invariant** $E[u, v] => \neg u.e \vee \neg v.e$

Conclusion: $u.e \mapsto \neg u.e$

The problem is to derive a control system (operating system os) that guarantees the behavior of the combined philosopher and control system (d*iners*) given by

[CM88, Sec.12.2]:

$$\textbf{invariant} \quad \neg(u.e \wedge v.e \wedge E[u,v]) \qquad \textbf{in } \mathit{user} \,[\!]\, \mathit{os} \qquad \text{(dn1)}$$

$$u.h \mapsto u.e \qquad \textbf{in } \mathit{user} \,[\!]\, \mathit{os} \qquad \text{(dn2)}$$

Note: We have, here and in the following, eased the notation by assuming implicit universal quantification over any free variables in program properties. That is, e.g., (dn2) should read:

$$\forall u :: u.h \mapsto u.e$$

1.2 The Diners Algorithm

We now describe the (final) algorithm presented in [CM88, Chap.12].

1.2.1 Informal Description

With each edge we associate a fork, which can be clean or dirty, and a request token. Initially all forks are dirty and all philosophers are thinking. Each fork is held by the owner with the lower number and each request token by the owner with the higher number.

A philosopher may eat only when he is hungry and has all his forks and none of them are dirty and requested by the other owner. All the forks get dirty when he starts eating. If he is hungry and miss a fork and has the corresponding request token, he sends this to the other owner of the fork. If he is not eating and has received a request for a fork, which is dirty, then he sends this fork to the other owner. A fork is cleaned when it is passed to the other owner.

1.2.2 Formal Specification of the Control System

Forks and requests are represented as shared data structures $fork[u,v]$ and $rf[u,v]$. The value of $fork[u,v]$ ($rf[u,v]$) indicates who currently holds the fork (request). The status (clean/dirty) of each fork is recorded in the shared boolean matrix $clean[u,v]$. The following UNITY specification of the control system (o*s of* d*inners*) is taken from [CM88, Sec.12.8].

$$\textbf{constant} \quad u.t \qquad \textbf{in } \mathit{os} \qquad \text{(odn1)}$$

$$\textbf{stable} \quad u.e \qquad \textbf{in } \mathit{os} \qquad \text{(odn2)}$$

$$\textbf{invariant} \quad G' \text{ is acyclic} \qquad \textbf{in } \mathit{os} \qquad \text{(odn7)}$$

$$\textbf{invariant} \quad u.e \wedge E[u,v] \Rightarrow (fork[u,v] = u) \wedge \neg clean[u,v] \qquad \textbf{in } \mathit{os} \qquad \text{(odn10)}$$

$$(fork[u,v] = v) \wedge clean[u,v] \textbf{ unless } v.e \qquad \textbf{in } os \quad \text{(odn11)}$$
$$(fork[u,v] = u) \wedge \neg clean[u,v] \textbf{ unless }$$
$$(fork[u,v] = v) \wedge clean[u,v] \qquad \textbf{in } os \quad \text{(odn12)}$$
$$\textbf{invariant} \quad (fork[u,v] = u) \wedge clean[u,v] \Rightarrow u.h \qquad \textbf{in } os \quad \text{(odn13)}$$
$$\textbf{invariant} \quad (fork[u,v] = u) \wedge (rf[u,v] = u) \Rightarrow v.h \qquad \textbf{in } os \quad \text{(odn14)}$$
$$(rf[u,v] = u) \textbf{ unless } sendreq[u,v] \qquad \textbf{in } os \quad \text{(odn15)}$$
$$sendreq[u,v] \textbf{ ensures } (rf[u,v] = v) \qquad \textbf{in } os \quad \text{(odn16)}$$
$$(fork[u,v] = u) \textbf{ unless } sendfork[u,v] \qquad \textbf{in } os \quad \text{(odn17)}$$
$$sendfork[u,v] \textbf{ ensures } (fork[u,v] = v) \qquad \textbf{in } os \quad \text{(odn18)}$$
$$(u.h \wedge u.mayeat) \textbf{ ensures } \neg(u.h \wedge u.mayeat) \qquad \textbf{in } os \quad \text{(odn19)}$$

where

$$sendreq[u,v] \equiv (fork[u,v] = v) \wedge (rf[u,v] = u) \wedge u.h \tag{4}$$
$$sendfork[u,v] \equiv (fork[u,v] = u) \wedge (rf[u,v] = u) \wedge \neg u.e \tag{5}$$
$$u.mayeat \equiv \forall v : E[u,v] : \; (fork[u,v] = u) \wedge (clean[u,v] \vee (rf[u,v] = v)) \tag{6}$$

Property (odn7) refers to G', which is the directed graph obtained from e by directing edge u-v from u to v if and only if u has higher priority than v. The priority is defined by:

$$(prior[u,v] = u) \equiv ((fork[u,v] = u) = clean[u,v])$$

The acyclicity of G' is preserved as long as rules 1 and 2 [CM88, p.297] are obeyed:

1. The direction of an edge is changed only when a process on which the edge is incident changes state from hungry to eating.
2. All edges incident on an eating process are directed toward it; equivalently an eating process is lower in order than all its neighbors.

The second rule is obviously obeyed since a philosopher holds all his forks when he eats and they all are dirty (odn10). The first rule must similarly be obeyed due to (odn10-odn12). (odn7) is thus only needed to ensure proper initialization.

In [CM88, Chap.12] it has been proved that an implementation fulfilling this specification will implement a correct solution to the problem. One such implementation is the shared memory UNITY program given in [CM88, Sec.12.9] (see below).

Program os
always
$\forall u :: (u.mayeat = \forall v : E[u,v] : (fork[u,v] = u) \wedge (clean[u,v] \vee (rf[u,v] = v)))$
$\forall u, v : E[u,v] : sendreq[u,v] = ((fork[u,v] = v) \wedge (rf[u,v] = u) \wedge u.h)$
$\forall u, v : E[u,v] : sendfork[u,v] = ((fork[u,v] = v) \wedge \neg clean[u,v] \wedge (rf[u,v] = u) \wedge \neg u.e)$
initially
$\forall u :: u.dine = t$
$\forall u, v :: \neg clean[u,v]$
$\forall u, v : u < v : (fork[u,v] = u) \wedge (rf[u,v] = v)$
assign
$\langle [\!] u :: u.h \wedge u.mayeat \;\rightarrow\; u.dine := e \parallel (\parallel v : E[u,v] : clean[u,v] := false) \;\rangle$
$[\!]$
$\langle [\!] (u,v) :: sendreq[u,v] \;\rightarrow\; rf[u,v] := v \;\rangle$
$[\!]$
$\langle [\!] (u,v) :: sendfork[u,v] \;\rightarrow\; fork[u,v], clean[u,v] := v, true \;\rangle$
end

Figure 2: Diners Shared Memory UNITY Implementation.

1.3 Shared Memory UNITY Implementation

Figure 2 shows Chandy and Misra's UNITY implementation of the solution to the diners problem using a slightly different syntax than [CM88].

1.4 Towards a Distributed Implementation

Chandy and Misra [CM88, p.311] suggest the following implementation scheme for a distributed memory system:

> The reader may replace the shared variables $fork[u,v]$, $clean[u,v]$, and $rf[u,v]$ in program os by variables local to processes u, v and channels, in both directions, between u and v. Implement $fork[u,v]$ and $rf[u,v]$ as tokens that are at the processes or in the channels between them. Implement $clean[u,v]$ as an attribute to the token $fork[u,v]$.

As we replace the shared variables by channels, messages and local variables we also decompose os into subprocesses. For a network of N philosophers, we get:

$$os \;=\; server_0 \;[\!]\; \ldots \;[\!]\; server_N$$

With the above scheme a server with n neighbors will have $2n$ incoming and $2n$ outgoing point-to-point unidirectional channels with buffer capacity one. For a uniform network where each node has degree d, this amounts to $2dN$ channels.

On e.g., a physically distributed system, it is very reasonable to assume that the cost of maintaining n point-to-point channels with buffer capacity k exceeds the cost of maintaining one channel with buffer capacity nk. This indicates that a scheme, where messages from neighbors are merged on a single Multiple-Sender-Single-Receiver (MSSR) channel for each server, is preferable.

Furthermore it seems reasonable to assume that the philosophers are distributed as well, i.e. *user* is decomposed into subprocesses:

$$user \quad = \quad client_0 \ [\!] \ \ldots \ [\!] \ client_N$$

where each *client* corresponds to a philosopher. The *client* and its *server* may or may not reside within the same address space on the distributed system, so to obtain full generality, we also replace the shared variable *u.dine* by local variables, messages and channels.

Note: By this replacement we actually modify the problem specification, since we alter the interface between *user* and *os*. Alternatively *user* could have been encapsulated to obtain a suitable interface to the distributed control system.

Our solution idea is illustrated for a small network in figure 3. Channels are represented by solid lines ending in triangles for MSSR channels and arrows for point-to-point channels.

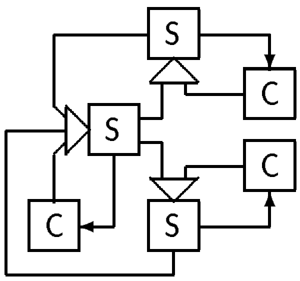

Figure 3: A network with three philosophers not fully connected.

2 Communication via Channels

Before we go into the details of distributing the shared variables, it is important to characterize the means of communication in CC++, since this is what guides the refinement.

2.1 Channels in CC++

A number of communication and synchronization primitives are provided as library routines in CC++. A thorough discussion of these is beyond the scope of this paper and may be found in [Siv93]. In the following passages we give a short description of the communication primitives used in this context, i.e. a subset of the routines involving message passing via channels.

A channel in CC++ is very general; It may be dynamic, multi-directional, have any buffer capacity and any number of senders and receivers. The full generality is however seldom used. In our example we will use static, unidirectional, point-to-point and MSSR (Multiple Sender Single Receiver) channels with unbounded buffer capacity.

Each channel is essentially a first-come-first-serve strongly typed FIFO queue of messages. Communicating entities know the address of the channel, but may only access the message queue via public member functions. We will only describe and use a small subset of these functions.

Given a message `m` of type `Msg` and a channel `C` of messages of type `Msg`.

`C.nonblockingSend(m)` Execution of this library routine is atomic. The message `m` is transferred directly to the receiver on channel `C`, if the receiver is ready. Otherwise the message is appended to the message queue for channel `C`.

`C.blockingReceive(&m)` The blocking receive suspends until a message is available from channel `C`. The message is then transferred to the receiver (by storing it at location `&m`) and the call terminates.

In figure 4 we show an example of use and declaration of a channel `C` for messages of type `int`. The main program contains a declaration of the shared integer channel `C` followed by two spawn statements. A spawn statement creates a new thread of control, which executes in parallel with its parent-thread (some languages call this a fork). The spawn statements in main invoke an instance of the producer and an instance of the consumer connected via the shared integer channel `C`. The producer instance simply sends the message `2` on channel `C`. The consumer is slightly more complex; it must inform the channel object where to store the incoming message before a message can be received. Here we see another CC++ extension, the keyword `sync`. A sync variable is a delayed constant; it may be assigned only once, but it need not be at the time of declaration. Such variables provide a means of synchronization since reading an undefined sync variable results in suspension until the variable has been defined. In our example the blocking receive will suspend until the sync pointer to the integer message has been initialized, which happens as soon as a message is

```
#include "Channel.h"

void producer(Channel<int>& outC, int m)
{ outC.nonblockingSend(m);
}

void consumer(Channel<int>& inC)
{ int * sync m;
  inC.blockingReceive(&m);
  int n = *m;
}

int main()
{ Channel<int> C;
  spawn producer(C,2);
  spawn consumer(C);
}
```

Figure 4: Channel Communication in CC++.

available. The "naked" integer message (n) is then obtained using the message pointer `m`.

2.2 Corresponding UNITY Channels

We model the CC++ channels in UNITY as shared lists (or sequences) of messages with head, tail, empty and append functions described below:

hd(C) matches the head of the message queue `C`.

tl(C) matches message queue `C` without the first message.

empty(C) is true when no messages are available.

$C^\frown(m)$ appends message `m` to message queue `C`.

The last operation corresponds to the CC++ nonblocking send, whereas the blocking receive is modelled using the head, tail and empty operations. To illustrate the relation to the CC++ channel operations, we show how the previous example translates to UNITY in figure 5.

The CC++ program has three threads of control; one for the main program, one for the producer instance, and one for the consumer instance. We model this in UNITY as a composition of three programs:

$$system \equiv main \; [\!] \; producer(C, 2) \; [\!] \; consumer(C)$$

Parallel execution of the threads is modelled by interleaving the actions of the components. To ensure sequential execution of the statements of each component we introduce program counters. Execution begins when the counter is one and terminates when (if ever) the counter is zero.

```
Program main
declare
    C : sequence of integer
    m.ctr, p.ctr, c.ctr : number
initially
    empty(C) ∧ m.ctr = 1 ∧ p.ctr = c.ctr = 0
assign
    ⟨m.ctr = 1  →  m.ctr,p.ctr := 2,1 ⟩
[]
    ⟨m.ctr = 2  →  m.ctr,c.ctr := 0,1 ⟩
end

Program producer(outC : reference,m : integer)
assign
    ⟨p.ctr = 1  →  p.ctr,outC := 0,outC⌢(m) ⟩
end

Program consumer(inC : reference)
declare
    m,n : integer
assign
    ⟨c.ctr = 1 ∧ ¬empty(inC)  →  c.ctr,m,inC := 2,hd(inC),tl(inC) ⟩
[]
    ⟨c.ctr = 2  →  c.ctr,n := 0,m ⟩
end
```

Figure 5: Channel Communication in UNITY.

The main program declares the channel and the counters and enables execution of the producer and the consumer actions by setting the corresponding counters to one. The producer appends the message (2) to the queue C and terminates. The consumer is stuck until the queue becomes non-empty upon which it consumes the first message and updates its counter. The second statement of the consumer simply copies m to n and terminates the consumer program (since C is a simple sequence in UNITY there is no need for pointers).

2.2.1 Suspension

Since all UNITY actions are atomic, we must simulate the receiver's suspension on an empty queue by special means. E.g., as in figure 5, by introducing a program counter, so once a receive action is initiated no other action is enabled until a message has been received.

2.3 Predicates on UNITY Channels

To be able to reason about the contents of the message queue we introduce a few general predicates on channels. Let u identify a channel, then:

$$\begin{array}{rcl} u.sX & \equiv & \text{the number of } X \text{ messages sent on the channel.} \\ u.rX & \equiv & \text{the number of } X \text{ messages received on the channel.} \\ u.prec(X,Y) & \equiv & \text{the first } Y \text{ message currently in the channel} \\ & & \text{is preceded by an } X \text{ message.} \\ u.recvd(X) & \equiv & \text{that particular message is removed from the head} \\ & & \text{of the channel buffer.} \end{array}$$

3 From Shared to Distributed State

In this section we explain how the shared variables are replaced by channels, messages and local variables.

3.1 Interface Between Client and Server

We replace *u.dine* by variables *u.cdine* local to $client_u$ and *u.sdine* local to $server_u$, messages *want* and *done* on $server_u$'s input channel, and messages *eat* on $client_u$'s input channel. We introduce the notation:

$$\begin{array}{rcl} u.ct \equiv (u.cdine = t) & \text{and} & u.st \equiv (u.sdine = t) \\ u.ch \equiv (u.cdine = h) & \text{and} & u.sh \equiv (u.sdine = h) \\ u.ce \equiv (u.cdine = e) & \text{and} & u.se \equiv (u.sdine = e) \end{array}$$

$$\begin{array}{rcl} u.W & \equiv & \text{there is a } want \text{ message in } server_u\text{'s input channel} \\ u.D & \equiv & \text{there is a } done \text{ message in } server_u\text{'s input channel} \\ u.E & \equiv & \text{there is an } eat \text{ message in } client_u\text{'s input channel} \end{array}$$

3.1.1 Client-Server Interface Restrictions

To ensure that the interface is used appropriately we formalize the restrictions:

$$\begin{array}{lll}
\textbf{constant} & u.cdine & \textbf{in } P,\ client_u \not\subseteq P \\
\textbf{constant} & u.sdine & \textbf{in } P,\ server_u \not\subseteq P \\
\textbf{constant} & u.W & \textbf{in } P,\ (client_u, server_u) \not\subseteq P \\
\textbf{constant} & u.D & \textbf{in } P,\ (client_u, server_u) \not\subseteq P \\
\textbf{constant} & u.E & \textbf{in } P,\ (client_u, server_u) \not\subseteq P
\end{array}$$

The notation **in** P, $(p, \ldots) \not\subseteq P$ denotes that none of $(p, \ldots)$ is equivalent to P nor a component of P. Thus the above formulae defines e.g., $u.cdine$ to be local to $client_u$ and $u.E$ to be a local predicate of $client_u \| server_u$.

We may now express $u.t$, $u.h$ and $u.e$ in terms of the messages and the distributed variables of either $client_u$ or $server_u$:

$$\begin{array}{rcll}
u.t & \equiv & u.ct & \\
 & \equiv & (u.st \wedge \neg u.W) \vee (u.se \wedge u.D \wedge \neg u.E) & (7) \\
u.h & \equiv & u.ch \wedge \neg u.E & \\
 & \equiv & u.sh \vee (u.st \wedge u.W) \vee (u.se \wedge u.D \wedge u.W) & (8) \\
u.e & \equiv & u.ce \vee (u.ch \wedge u.E) & \\
 & \equiv & u.se \wedge \neg u.D & (9)
\end{array}$$

given that the following proof obligations are satisfied:

$$\begin{array}{llll}
\textbf{invariant} & u.ct = (u.st \wedge \neg u.W) \vee (u.se \wedge u.D \wedge \neg u.W) & \textbf{in } user \| os & (10) \\
\textbf{invariant} & u.ch \wedge \neg u.E = u.sh \vee (u.st \wedge u.W) \vee (u.se \wedge u.D \wedge u.E) & \textbf{in } user \| os & (11) \\
\textbf{invariant} & u.ce \vee (u.ch \wedge u.E) = u.se \wedge \neg u.D & \textbf{in } user \| os & (12)
\end{array}$$

3.2 The User Specification

In the distributed version we assume that the client sends or receives a message every time it changes state. Thus a transition from $u.ct$ to $u.ch$ is reflected by sending message *want*, transition $u.ch$ to $u.ce$ by receiving message *eat*, and finally transition $u.ce$ to $u.ct$ by sending message *done*.

3.2.1 Translated User Specification

The translated user specification is derived from (udn1-udn4) using relations (7-9) from above.

$u.ct$ **unless** $u.ch \wedge \neg u.E$	**in** *user* (udn1')
stable $(u.ch \wedge \neg u.E)$	**in** *user* (udn2')
$u.ce \vee (u.ch \wedge u.E)$ **unless** $u.ct$	**in** *user* (udn3')
Conditional Property	**in** *user* (udn4')

Hypothesis: **invariant** $E[u,v] => \neg(u.ce \vee (u.ch \wedge u.E)) \vee$
$\neg(v.pe \vee (v.ph \wedge v.E))$

Conclusion: $(u.ch \wedge u.E) \vee u.ce \mapsto \neg(u.ch \wedge u.E) \wedge \neg u.ce$

The functionality of *user* is not quite obvious from the translated specification. However, a much clearer specification may be derived in a single refinement step.

3.2.2 Refined User Specification

Initially, the user is thinking and has neither sent nor received any messages. The user remains thinking unless it gets hungry and sends a *want* message. It remains hungry unless it receives an *eat* message and starts eating. It remains eating unless it sends a *done* message and starts thinking. The user never sends *eat* messages, and cannot recall already sent *want* and *done* messages. It does indeed start eating if it is hungry and an *eat* message is available. It does not eat forever; eventually it sends a *done* message and returns to thinking.

initially $u.ct \wedge (u.sW = u.rE = u.sD = 0)$	**in** $client_u$	(13)
invariant $(u.ct \wedge (u.sW = u.rE = u.sD)) \vee$ $(u.ch \wedge (u.sW = u.rE + 1 = u.sD + 1)) \vee$ $(u.ce \wedge (u.sW = u.rE = u.sD + 1))$	**in** $client_u$	(14)
invariant $u.D \wedge u.W \Rightarrow u.prec(D, W)$	**in** $client_u$	(15)
stable $\neg u.E$	**in** $client_u$	(16)
stable $u.W$	**in** $client_u$	(17)
stable $u.D$	**in** $client_u$	(18)
$u.ct$ **unless** $u.ch \wedge u.W$	**in** $client_u$	(19)
$u.ch$ **unless** $u.ch \wedge u.E$	**in** $client_u$	(20)
$u.ch \wedge u.E$ **ensures** $u.ce$	**in** $client_u$	(21)
$u.ce$ **ensures** $u.ct \wedge u.D$	**in** $client_u$	(22)

Note: $a = b = c$ denotes $(a = b) \wedge (b = c)$ and not $(a = b) = c$.

Locality and Compositionality Since the other clients have no access to $u.cdine$, $client_u$'s input channel or $server_u$'s input channel, the above properties hold trivially for $client_v$, $v \neq u$. Properties (13-22) may thus be viewed equally well as properties of $client_u$ or $user$.

3.2.3 Correctness of Refinement

Viewing properties (13-22) as properties of $user$ we may derive (udn1'-udn4') as follows:

- (udn1') follows from (19) and **invariant** $u.W \Rightarrow \neg u.E$ of the composed system.
- (udn2') follows from the definition of **stable** and conjunction on (20) and (16).
- (udn3') follows from cancellation on **unless** properties from (21) and (22) and weakening.
- (udn4') follows from (21) and (22).

So given that we can prove **invariant** $u.W \Rightarrow \neg u.E$ **in** $user [\!] os$, we may replace the translated user specification with the more intuitive specification.

3.3 Interface Between Servers

As for $u.dine$, we replace $fork[u, v]$, $clean[u, v]$ and $rf[u, v]$ by arrays $f[v]$, $clean[v]$ and $r[v]$ local to each server and channels between the servers.

Each pair of neighbors u and v share exactly one fork and one request. We introduce the notation:

$$\begin{aligned}
u.f.v &\equiv \text{ } u \text{ holds the fork shared with } v \ (\equiv u.f[v]) \\
u.fc.v &\equiv \text{ the fork is in the channel from } u \text{ to } v \\
u.r.v &\equiv \text{ } u \text{ holds the request shared with } v \ (\equiv u.r[v]) \\
u.rc.v &\equiv \text{ the request is in the channel from } u \text{ to } v \\
u.clean.v &\equiv \text{ the fork is clean } (\equiv u.clean[v] \text{ if } u \text{ holds the fork})
\end{aligned}$$

The fork is always clean while in the channel:

$$\textbf{invariant} \quad u.fc.v \Rightarrow u.clean.v \qquad \textbf{in } server_u [\!] server_v$$

3.3.1 Server-Server Interface Restrictions

The locality of the variables and predicates can be expressed as:

$$\begin{array}{lll}
\textbf{constant} & u.f[v] & \textbf{in } P,\ server_u \not\subseteq P \\
\textbf{constant} & u.r[v] & \textbf{in } P,\ server_u \not\subseteq P \\
\textbf{constant} & u.clean[v] & \textbf{in } P,\ server_u \not\subseteq P \\
\textbf{constant} & u.f.v & \textbf{in } P,\ server_u \not\subseteq P \\
\textbf{constant} & u.r.v & \textbf{in } P,\ server_u \not\subseteq P \\
\textbf{constant} & u.clean.v & \textbf{in } P,\ server_u \not\subseteq P \\
\textbf{constant} & u.fc.v & \textbf{in } P,\ (server_u, server_v) \not\subseteq P \\
\textbf{constant} & u.rc.v & \textbf{in } P,\ (server_u, server_v) \not\subseteq P
\end{array}$$

We interpret $fork[u, v]$ and $rf[u, v]$ and $clean[u, v]$ in terms of the new notation:

$$(fork[u, v] = u) \equiv u.f.v \vee v.fc.u \tag{23}$$

$$(rf[u, v] = u) \equiv u.r.v \vee v.rc.u \tag{24}$$

$$\begin{aligned} clean[u, v] \equiv\ & v.fc.u \vee (v.f.u \wedge v.clean.u) \vee \\ & u.fc.v \vee (u.f.v \wedge u.clean.v) \end{aligned} \tag{25}$$

Consistency proof obligations:

$$u.f.v \vee v.fc.u \equiv \neg(v.f.u \vee u.fc.v) \tag{26}$$

$$u.r.v \vee v.rc.u \equiv \neg(v.r.u \vee u.rc.v) \tag{27}$$

are trivial given system invariants

$$\textbf{invariant} \quad E[u, v] \Rightarrow ExactlyOneOf(u.f.v, v.f.u, u.fc.v, v.fc.u) \tag{28}$$

$$\textbf{invariant} \quad E[u, v] \Rightarrow ExactlyOneOf(u.r.v, v.r.u, u.rc.v, v.rc.u) \tag{29}$$

where *ExactlyOneOf* has the obvious definition that exactly one of its argument predicates is true. For non-neighbors v, the channels remain empty all time and we define $u.f[v]$ to be true and $u.r[v]$ and $u.clean[v]$ false:

$$\begin{aligned} \textbf{invariant} \quad \neg E[u, v] \Rightarrow (\ & u.f.v \wedge \neg u.clean.v \wedge \neg u.r.v \wedge \\ & v.f.u \wedge \neg v.r.u \wedge \neg v.clean.u \wedge \\ & NoneOf(u.fc.v, u.rc.v, v.fc.u, v.rc.u)) \ \textbf{ in } os \end{aligned} \tag{30}$$

where *NoneOf* has the obvious definition that none of the argument predicates are true.

3.4 Translation of Predicates

For convenience we introduce the shorthands:

$$\begin{aligned}
u.sendreq.v &\equiv \neg u.f.v \wedge u.r.v \wedge u.h' && (31)\\
u.sendfork.v &\equiv u.f.v \wedge \neg u.clean.v \wedge (u.r.v \vee v.rc.u) \wedge \\
& \quad (\neg u.se \vee (u.se \wedge u.D)) && (32)\\
u.mayeat' &\equiv \forall v : E[u,v] : (u.f.v \wedge (u.clean.v \vee v.r.u)) \vee v.fc.u && (33)\\
u.h' &\equiv u.sh \vee (u.st \wedge u.W) \vee (u.se \wedge u.D \wedge u.W) && (34)
\end{aligned}$$

and translate some predicates which occur in several of the properties of os:

$$\begin{aligned}
(fork[u,v] = u) \wedge \neg clean[u,v] &\equiv u.f.v \wedge \neg u.clean.v && (35)\\
(fork[u,v] = v) \wedge clean[u,v] &\equiv (v.f.u \wedge v.clean.u) \vee u.fc.v && (36)\\
(fork[u,v] = v) \wedge (rf[u,v] = u) &\equiv (v.f.u \vee u.fc.v) \wedge u.r.v \\
&\equiv \neg u.f.v \wedge u.r.v && (37)\\
(fork[u,v] = u) \wedge (rf[u,v] = u) &\equiv \neg(u.fc.v \vee v.f.u \vee u.rc.v \vee v.r.u) && (38)\\
sendreq[u,v] &\equiv u.sendreq.v && (39)\\
sendfork[u,v] &\equiv u.sendfork.v && (40)\\
u.mayeat &\equiv u.mayeat' && (41)
\end{aligned}$$

3.4.1 Correctness of Translation of Predicates

We prove the correctness of the predicate translation using earlier derived invariants plus:

$$\begin{aligned}
&\textbf{invariant} \quad u.rc.v \Rightarrow \neg(u.f.v \vee v.fc.u) \quad && \textbf{in } os && (42)\\
&\textbf{invariant} \quad v.fc.u \Rightarrow \neg u.r.v \quad && \textbf{in } os && (43)
\end{aligned}$$

That is, if the request is in the channel from u to v, the fork is neither at u nor in the channel from v to u. Similarly, if the fork is in the channel from v to u, the request is not at u (it might be in the channel though from v to u).

- (35) follows from (23), (25), and invariant (28).
- (36) follows from (23) and (25).
- (37) follows from (23), (24), (26), (42) and (43).
- (38) follows from (23), (24), (26) and (27).
- (39) follows from (4), (37), (8), (31) and (34).

- (40) follows from (5), (35), (24), (9), and (32).
- (41) follows from (6), (36), (37), and (33).

3.5 Extra Safety Properties

We summarize the extra safety properties discussed during the above introduction of a distributed state, before we continue with the translation of the *os* safety properties.

invariant	$u.ct = \ (u.st \wedge \neg u.W) \vee (u.se \wedge u.D \wedge \neg u.W)$	**in** $user \| os$
invariant	$u.ch \wedge \neg u.E = \ u.sh \vee (u.st \wedge u.W) \vee (u.se \wedge u.D \wedge u.W)$	**in** $user \| os$
invariant	$u.ce \vee (u.ch \wedge u.E) = u.se \wedge \neg u.D$	**in** $user \| os$
invariant	$u.W \Rightarrow \neg u.E$	**in** $user \| os$
invariant	$E[u, v] \Rightarrow \mathit{ExactlyOneOf}(u.f.v, v.f.u, u.fc.v, v.fc.u)$	**in** *os*
invariant	$E[u, v] \Rightarrow \mathit{ExactlyOneOf}(u.r.v, v.r.u, u.rc.v, v.rc.u)$	**in** *os*
invariant	$\neg E[u, v] \Rightarrow (\ u.f.v \wedge \neg u.clean.v \wedge \neg u.r.v \wedge v.f.u \wedge \neg v.r.u \wedge \neg v.clean.u \wedge \mathit{NoneOf}(u.fc.v, u.rc.v, v.fc.u, v.rc.u))$	**in** *os*
invariant	$u.rc.v \Rightarrow \neg(u.f.v \vee v.fc.u)$	**in** *os*
invariant	$v.fc.u \Rightarrow \neg u.r.v$	**in** *os*

4 Safety

The safety properties of *os* (see page 4) are first translated and then refined to ease the implementation.

4.1 Translated Safety Properties

We here list the safety properties of *os* in terms of predicates on the distributed state:

constant $(u.st \wedge \neg u.W) \vee (u.se \wedge u.D \wedge \neg u.W)$	**in** *os*	(odn1')
stable $u.se \wedge \neg u.D$	**in** *os*	(odn2')
invariant G' is acyclic	**in** *os*	(odn7')
invariant $(u.se \wedge \neg u.D) \wedge E[u, v] \Rightarrow u.f.v \wedge \neg u.clean.v$	**in** *os*	(odn10')
$(v.f.u \wedge v.clean.u) \vee u.fc.v$ **unless** $(v.se \wedge \neg v.D)$	**in** *os*	(odn11')

$u.f.v \wedge \neg u.clean.v$ **unless** $(v.f.u \wedge v.clean.u) \vee u.fc.v$ **in** os (odn12')

invariant $(u.f.v \wedge u.clean.v) \vee v.fc.u \Rightarrow u.h'$ **in** os (odn13')

invariant $\neg(v.f.u \vee u.fc.v) \wedge \neg(v.r.u \vee u.rc.v) \Rightarrow v.h'$ **in** os (odn14')

$u.r.v \vee v.rc.u$ **unless** $u.sendreq.v$ **in** os (odn15')

$u.f.v \vee v.fc.u$ **unless** $u.sendfork.v$ **in** os (odn17')

We introduce an extra refinement step, since the properties that refer to messages on input channels as well as output channels are somewhat difficult to fulfill directly by an implementation.

4.2 Refinement of Safety Properties

Messages on input channels may only be observed and received when they have reached the head of the channel. Hence we distinguish this situation and use that in the refinement; e.g., for sending forks and requests we identify the intermediate states:

$$u.passrequest.v \equiv \neg u.f.v \wedge u.r.v \wedge u.sh$$
$$u.passfork.v \equiv (u.rcv(D) \wedge u.r.v) \vee (u.rcv(R.v) \wedge \neg u.clean.v \wedge \neg u.se)$$

where

$u.rcv(M) \equiv$ $server_u$ is ready to receive a message and
M is at the head of its input channel

This leads to the following refined properties

initially $(E[u,v] \wedge u>v) \Rightarrow \neg u.f.v \wedge u.r.v \wedge \neg u.clean.v$ **in** $server_u$ (44)

initially $(\neg E[u,v] \wedge u \leq v) \Rightarrow u.f.v \wedge \neg u.r.v \wedge \neg u.clean.v$ **in** $server_u$ (45)

invariant $u.se \Rightarrow u.f.v \wedge \neg u.clean.v$ **in** $server_u$ (46)

invariant $(u.f.v \wedge u.clean.v) \Rightarrow u.sh$ **in** $server_u$ (47)

invariant $\neg u.f.v \wedge \neg u.r.v \Rightarrow u.sh$ **in** $server_u$ (48)

invariant $u.rcv(D) \Rightarrow u.se$ **in** $user [\!] os$ (49)

invariant $u.rcv(R.v) \Rightarrow u.f.v$ **in** $user [\!] os$ (50)

invariant $u.E \Rightarrow \neg u.D$ **in** $user [\!] os$ (51)

$u.E$ **unless** $false$ **in** $server_u$ (52)

$\neg u.D$ **unless** $false$ **in** $server_u$ (53)

$\neg u.W$ **unless** $false$ **in** $server_u$ (54)

$u.D$ **unless** $u.rcv(D)$ **in** $server_u$ (55)

$u.W$ **unless** $u.rcv(W)$ **in** $server_u$ (56)

$u.st$ **unless** $u.rcv(W)$ **in** $server_u$ (57)

$u.se$ **unless** $u.rcv(D)$ **in** $server_u$ (58)

$u.rcv(D)$ **unless** $u.st$ **in** $server_u$ (59)

$u.rcv(W)$ **unless** $u.sh$ **in** $server_u$ (60)

$u.sh$ **unless** $u.se \wedge u.E$ **in** $server_u$ (61)

$v.fc.u$ **unless** $u.f.v \wedge u.clean.v$ **in** $server_u$ (62)

$v.f.u \wedge v.clean.u$ **unless** $v.se \wedge v.E$ **in** $server_v$ (63)

$u.f.v \wedge \neg u.clean.v$ **unless** $u.fc.v$ **in** $server_u$ (64)

$u.f.v$ **unless** $u.passfork.v$ **in** $server_u$ (65)

$v.rc.u$ **unless** $u.r.v$ **in** $server_u$ (66)

$u.r.v$ **unless** $u.passrequest.v$ **in** $server_u$ (67)

Locality and Compositionality Using the interface restrictions we may prove that the above properties are also properties of $user \,[]\, os$.

4.3 Correctness of Safety Refinement

Viewing properties (44-67) as properties of $user \,[]\, os$ we find:

- (odn10') follows from (46) and weakening.
- (odn11') follows from cancellation on (62) with u and v interchanged, and (63) followed by weakening using invariant (51).
- (odn12') follows from weakening (64).
- (odn13') follows from (42) and (28), (48) and (47) and weakening the rhs.
- (odn14') follows from (48) and weakening.
- (odn15') follows from cancellation on (66) and (67) and weakening.
- (odn17') follows from weakening (62), cancellation with (65) and weakening using invariants (49) and (50).

As explained page 4 (odn7) and hence (odn7') follows from (odn10-odn12) given that G' is acyclic initially. (odn7') is thus refined by (44) and (45), which guarantees acyclicity initially, and the properties which refine (odn10'-odn12').

Finally, (odn1')—and (odn2') as a subproof—follows from

- **stable** $(u.st \wedge \neg u.W) \vee (u.se \wedge u.D \wedge \neg u.W)$
  which follows from cancellation on

$$\begin{array}{ll} u.se \wedge u.D \wedge \neg u.W \textbf{ unless } u.st \wedge \neg u.W & \textbf{in } server_u \\ u.st \wedge \neg u.W \textbf{ unless } false & \textbf{in } server_u \end{array}$$

  where the first of these follows from conjunction on

$$\begin{array}{ll} \neg u.W \textbf{ unless } false & \textbf{in } server_u \\ u.st \textbf{ unless } u.rcv(W) & \textbf{in } server_u \end{array}$$

  and the second from conjunction on

$$\begin{array}{ll} \neg u.W \textbf{ unless } false & \textbf{in } server_u \\ u.se \wedge u.D \textbf{ unless } u.st & \textbf{in } server_u. \end{array}$$

  Finally this last property follows from conjunction on

$$\begin{array}{ll} u.se \textbf{ unless } u.rcv(D) & \textbf{in } server_u \\ u.D \textbf{ unless } u.rcv(D) & \textbf{in } server_u \\ u.rcv(D) \textbf{ unless } u.st & \textbf{in } server_u \end{array}$$

  given **invariant** $u.rcv(D) \Rightarrow u.se$ **in** $user [\!] os$.

- **stable** $\neg(u.st \wedge \neg u.W) \wedge \neg(u.se \wedge u.D \wedge \neg u.W)$.
  Since this can be rewritten to **stable** $u.W \vee u.sh \vee (u.se \wedge \neg u.D)$
  we derive it by cancellation on

$$\begin{array}{ll} u.W \textbf{ unless } u.sh & \textbf{in } server_u \\ u.sh \textbf{ unless } u.se \wedge \neg u.D & \textbf{in } server_u \\ u.se \wedge \neg u.D \textbf{ unless } false & \textbf{in } server_u, \end{array}$$

  where the first of these follows from cancellation on

$$\begin{array}{ll} u.W \textbf{ unless } u.rcv(W) & \textbf{in } server_u \\ u.rcv(W) \textbf{ unless } u.sh & \textbf{in } server_u \end{array}$$

  and the second by consequence weakening on

$$\begin{array}{ll} u.sh \textbf{ unless } u.se \wedge u.E & \textbf{in } server_u \\ \textbf{invariant } u.E \Rightarrow \neg u.D & \textbf{in } server_u \end{array}$$

  and the third is (odn2') which follows from conjunction on

$$\begin{array}{ll} u.se \textbf{ unless } u.rcv(D) & \textbf{in } server_u \\ \neg u.D \textbf{ unless } false & \textbf{in } server_u. \end{array}$$

5 Progress

5.1 Progress Properties for Composite Program

The progress properties expressed in predicates on the distributed state becomes:

$u.sendreq.v$ **ensures** $v.r.u \vee u.rc.v$	**in** os	(odn16')
$u.sendfork.v$ **ensures** $v.f.u \vee u.fc.v$	**in** os	(odn18')
$(u.h' \wedge u.mayeat')$ **ensures** $\neg(u.h' \wedge u.mayeat')$	**in** os	(odn19')

All lhs's in the above properties cover states with two pending messages on the input channel. The intended progress is that both of these messages be received prior to or simultaneously with the establishment of the rhs's. However, in our channel implementation messages can only be received one at a time, so we cannot prove these properties from our program.

A closer look at the proof of correctness for the final refinement step in [CM88, Sec.12.7.3] shows that we might replace (odn16), (odn18) and (odn19) by the following properties of the composite system. (See [Bin93] for a completed proof of correctness for the last refinement step in [CM88, Sec.12.7.3] and the full proofs for the refinements in this report.):

$u.sendreq.v \mapsto v.r.u \vee u.rc.v$	**in** $user \,[\!]\, os$	(dn16')
$u.sendfork.v \mapsto v.f.u \vee u.fc.v$	**in** $user \,[\!]\, os$	(dn18')
$(u.h' \wedge u.mayeat') \mapsto \neg(u.h' \wedge u.mayeat')$	**in** $user \,[\!]\, os$	(dn19')

So we may replace (odn16'), (odn18'), and (odn19') in the os specification by any set of properties that refines properties (dn16'), (dn18'), and (dn19').

Refinement of Progress Properties We refine the properties in two steps. As described above progress from the lhs state to the rhs state requires at least two transitions with an intermediate state. The first refinement splits each property in (a) properties leading from the lhs state to an intermediate state, and (b) properties leading from there to the rhs state. The (b) properties are then further refined, whereas the more complex (a) properties are handled in a separate refinement step.

Locality and Compositionality Using the interface restrictions, it is rather easy (but boring) to prove that the properties of $server_u$ used in the following refinements also hold for $user \,[\!]\, os$.

5.2 Introducing an Intermediate State

5.2.1 Sending Requests

$$u.sendreq.v \mapsto v.r.u \vee u.rc.v \qquad \textbf{in } user \,\|\, os \qquad \text{(dn16')}$$

Strengthening the rhs to $u.rc.v$—since the request must pass through the channel before it is received by the other server—(dn16') follows by progress-safety-progress and transitivity on

$$\begin{array}{ll} u.h' \mapsto u.sh & \textbf{in } user \,\|\, os \\ \neg u.f.v \wedge u.r.v \textbf{ unless } u.passrequest.v & \textbf{in } server_u \\ u.passrequest.v \textbf{ ensures } u.rc.v & \textbf{in } server_u \end{array}$$

where the safety property follows from conjunction on

$$\begin{array}{ll} \neg u.f.v \textbf{ unless } u.rcv(F.v) & \textbf{in } server_u \\ u.r.v \textbf{ unless } u.passrequest.v & \textbf{in } server_u \end{array}$$

with **invariant** $v.fc.u \Rightarrow \neg u.r.v$ **in** $server_u$

5.2.2 Sending Forks

$$u.sendfork.v \mapsto v.f.u \vee u.fc.v \qquad \textbf{in } user \,\|\, os \qquad \text{(dn18')}$$

As for sending requests, we may strengthen the rhs to $u.fc.v$. Now using

$$\begin{array}{lll} \textbf{invariant} & u.se \Rightarrow u.f.v \wedge \neg u.clean.v & \textbf{in } server_u \\ \textbf{invariant} & u.D \Rightarrow u.se & \textbf{in } server_u \end{array}$$

the lhs of (dn18') may be rewritten to

$$\begin{aligned} u.sendfork.v = \; & \underbrace{u.r.v \wedge u.f.v \wedge \neg u.clean.v \wedge \neg u.se}_{(a)} \vee \underbrace{v.rc.u \wedge u.D}_{(b)} \vee \\ & \underbrace{u.r.v \wedge u.D}_{(c)} \vee \underbrace{v.rc.u \wedge u.f.v \wedge \neg u.clean.v \wedge \neg u.se}_{(d)} \end{aligned}$$

Case (a) never occurs if we assume that the server only holds both a dirty fork and the corresponding request while it is eating. For (b) both messages are on the same channel and must be received one at a time leading to either (c) or

(d). Which suggests the following refinement for sending forks:

$$
\begin{array}{ll}
\textbf{invariant} \quad u.f.v \wedge u.r.v \wedge \neg u.clean.v \Rightarrow u.se & \textbf{in } server_u \\
b \mapsto c \vee d & \textbf{in } user [] os \\
c \mapsto u.passfork.v & \textbf{in } user [] os \\
d \mapsto u.passfork.v & \textbf{in } user [] os \\
u.passfork.v \mapsto u.fc.v & \textbf{in } user [] os
\end{array}
$$

The last progress property follows from cancellation on

$$
\begin{array}{ll}
u.rcv(D) \wedge u.r.v \textbf{ ensures } u.fc.v & \textbf{in } server_u \\
u.rcv(R.v) \wedge u.st \textbf{ ensures } u.fc.v & \textbf{in } server_u \\
u.rcv(R.v) \wedge \neg u.clean.v \wedge u.sh \textbf{ ensures } u.fc.v & \textbf{in } server_u
\end{array}
$$

We also refine case (b), but save the others for the next refinement step.

Case (b)

$$
\begin{array}{l}
v.rc.u \ \wedge u.D \mapsto \\
\qquad (v.rc.u \wedge u.f.v \wedge \neg u.clean.v \wedge \neg u.se) \vee (u.r.v \wedge u.D) \textbf{ in } user [] os
\end{array}
$$

follows from cancellation on

$$
\begin{array}{ll}
v.rc.u \wedge u.D \mapsto \ (u.rcv(R.v) \wedge u.D) \vee (v.rc.u \wedge u.rcv(D)) & \textbf{in } user [] os \\
u.rcv(R.v) \wedge u.D \textbf{ ensures } u.r.v \wedge u.D & \textbf{in } server_u \\
u.rcv(D) \wedge v.rc.u \textbf{ ensures } v.rc.u \wedge u.f.v \wedge \neg u.clean.v \wedge \neg u.se & \textbf{in } server_u
\end{array}
$$

where each of these progress properties are derived below

- the first from transitivity and progress-safety-progress on

$$
\begin{array}{lll}
u.D \mapsto u.rcv(D) & & \textbf{in } user [] os \\
u.rcv(D) \textbf{ ensures } u.st & & \textbf{in } server_u \\
v.rc.u \wedge u.D \textbf{ unless} & (u.rcv(R.v) \wedge u.D) \vee & \\
 & (v.rc.u \wedge u.rcv(D)) & \textbf{in } server_u
\end{array}
$$

  with the safety property following from conjunction on

$$
\begin{array}{ll}
v.rc.u \textbf{ unless } u.rcv(R.v) & \textbf{in } server_u \\
u.D \textbf{ unless } u.rcv(D) & \textbf{in } server_u
\end{array}
$$

- the second follows from consequence weakening on the conjunction of

$$
\begin{array}{ll}
u.D \textbf{ unless } u.rcv(D) & \textbf{in } server_u \\
u.rcv(R.v) \wedge (u.se \vee u.clean.v) \textbf{ ensures } u.r.v & \textbf{in } server_u
\end{array}
$$

- and the third from consequence weakening on the conjunction of

$$\begin{array}{ll} v.rc.u \ \textbf{unless} \ u.rcv(R.v) & \textbf{in } server_u \\ u.rcv(D) \wedge \neg u.r.v \ \textbf{ensures} \ u.st \wedge u.f.v \wedge \neg u.clean.v & \textbf{in } server_u \end{array}$$

5.2.3 Towards Eating

$$(u.h' \wedge u.mayeat') \mapsto \neg(u.h' \wedge u.mayeat') \qquad \textbf{in } user \| os \qquad \text{(dn19')}$$

Choosing the intermediate state to be $u.sh \wedge u.forksready$ where

$$u.forksready \equiv \forall v : E[u,v] : u.f.v \wedge (u.clean.v \vee v.r.u)$$

we may refine (dn19') by

$$\begin{array}{ll} u.h' \mapsto u.sh & \textbf{in } user \| os \\ u.mayeat' \mapsto u.forksready \vee \neg(u.h' \wedge u.mayeat') & \textbf{in } user \| os \\ u.sh \ \textbf{unless} \ \neg(u.h' \wedge u.mayeat') & \textbf{in } server_u \\ u.sh \wedge u.forksready \mapsto \neg(u.h' \wedge u.mayeat') & \textbf{in } user \| os \end{array}$$

where the safety property follows from consequence weakening on

$$\begin{array}{ll} u.sh \ \textbf{unless} \ u.se \wedge u.E & \textbf{in } server_u \\ \textbf{invariant} \quad u.E \Rightarrow \neg u.D & \textbf{in } user \| os \end{array}$$

and the last progress property follows from transitivity on

$$\begin{array}{ll} \textbf{invariant} \quad u.sh \wedge u.cleanornoreq \Rightarrow u.starteat & \textbf{in } server_u \\ u.starteat \ \textbf{ensures} \ u.se \wedge u.E & \textbf{in } server_u \end{array}$$

where $u.cleanornoreq \equiv \forall v : E[u,v] : u.f.v \wedge (u.clean.v \vee \neg u.r.v)$ is obviously implied by $u.forksready$.

5.3 Reaching the Intermediate State

This refinement step involves the properties describing the progress from the original lhs to the intermediate state as defined above.

5.3.1 Reaching the Hungry State

$$u.h' \mapsto u.sh \qquad \textbf{in } user \| os$$

follows from cancellation and transitivity on

$$\begin{array}{ll} u.W \mapsto u.rcv(W) & \textbf{in } user \,\|\, os \\ u.rcv(W) \wedge u.miss = 0 \textbf{ ensures } u.sh \wedge u.starteat & \textbf{in } server_u \\ u.rcv(W) \wedge u.miss \neq 0 \textbf{ ensures } u.sh \wedge u.passreqs & \textbf{in } server_u \end{array}$$

where $u.miss$ indicates the number of forks the server is missing, and $u.starteat$ and $u.passrequests$ are enabling predicates (local to $server_u$) to restrict the execution order of the actions.

5.3.2 Reaching the Passfork State

Case (c)

$$u.r.v \wedge u.D \mapsto u.passfork.v \qquad \textbf{in } user \,\|\, os$$

follows from $u.r.v \wedge u.D \Rightarrow u.r.v \wedge u.f.v \wedge u.D$ which follows from

$$\begin{array}{lll} \textbf{invariant} & u.D \Rightarrow u.se & \textbf{in } user \,\|\, os \\ \textbf{invariant} & u.se \Rightarrow u.f.v \wedge \neg u.clean.v & \textbf{in } server_u \end{array}$$

and from progress-safety-progress and weakening on

$$\begin{array}{ll} u.D \mapsto u.rcv(D) & \textbf{in } user \,\|\, os \\ u.r.v \wedge u.f.v \textbf{ unless } u.rcv(D) \wedge u.r.v & \textbf{in } server_u \end{array}$$

where the safety property follows by weakening the conjunction of

$$\begin{array}{ll} u.r.v \textbf{ unless } u.passrequest.v & \textbf{in } server_u \\ u.f.v \textbf{ unless } u.passfork.v & \textbf{in } server_u \end{array}$$

given **invariant** $u.rcv(R.v) \Rightarrow u.f.v$ **in** $server_u$

Case (d)

$$v.rc.u \wedge u.f.v \wedge \neg u.clean.v \wedge \neg u.se \mapsto u.passfork.v \qquad \textbf{in } user \,\|\, os$$

follows from transitivity and cancellation on

$$\begin{array}{ll} v.rc.u \wedge u.f.v \wedge \neg u.clean.v \wedge \neg u.se \mapsto & \\ \quad v.rc.u \wedge u.prec(R.v, D) \wedge u.f.v \wedge \neg u.clean.v \wedge \neg u.se & \textbf{in } user \,\|\, os \\ v.rc.u \wedge u.prec(R.v, D) \wedge u.f.v \wedge \neg u.clean.v \wedge \neg u.se \mapsto & \\ \quad (u.rcv(R.v) \wedge \neg u.clean.v \wedge \neg u.se) \vee & \\ \quad (v.rc.u \wedge u.prec(R.v, D) \wedge u.se) & \textbf{in } user \,\|\, os \\ v.rc.u \wedge u.prec(R.v, D) \wedge u.se \mapsto (u.rcv(D) \wedge u.r.v) & \textbf{in } user \,\|\, os \end{array}$$

where

- the first progress property follows from
  invariant $u.D \Rightarrow u.se$ **in** $user \,[\!]\, os$

- and the second follows from progress-safety-progress on

  $$\begin{array}{ll} v.rc.u \mapsto u.rcv(R.v) & \textbf{in } user \,[\!]\, os \\ v.rc.u \wedge u.prec(R.v, D) \wedge u.f.v \wedge \neg u.clean.v \wedge \neg u.se & \\ \quad \textbf{unless } (u.rcv(R.v) \wedge \neg u.clean.v \wedge \neg u.se) \vee & \\ \qquad\quad (v.rc.u \wedge u.prec(R.v, D) \wedge u.se) & \textbf{in } user \,[\!]\, os \end{array}$$

  where the safety property follows from conjunction with tautology $\neg u.se$ **unless** $u.se$ and weakening on

  $$\begin{array}{ll} v.rc.u \wedge u.prec(R.v, D) \wedge u.f.v \wedge \neg u.clean.v \textbf{ unless} & \\ \qquad\qquad\qquad (u.rcv(R.v) \wedge \neg u.clean.v) & \textbf{in } server_u \end{array}$$

  which again follows from conjunction and weakening on

  $$\begin{array}{ll} v.rc.u \textbf{ unless } u.rcv(R.v) & \textbf{in } server_u \\ u.prec(R.v, D) \textbf{ unless } u.rcv(R.v) & \textbf{in } server_u \\ u.f.v \wedge \neg u.clean.v \textbf{ unless } u.passfork.v & \textbf{in } server_u \end{array}$$

The third progress property follows from transitivity on

$$\begin{array}{ll} v.rc.u \wedge u.prec(R.v, D) \wedge u.se \mapsto u.rcv(R.v) \wedge u.se & \textbf{in } user \,[\!]\, os \\ u.rcv(R.v) \wedge u.se \textbf{ ensures} & \\ \quad (u.rcv(D) \wedge u.r.v) \vee (u.r.v \wedge u.se) & \textbf{in } server_u \\ u.r.v \wedge u.se \mapsto (u.rcv(D) \wedge u.r.v) & \textbf{in } user \,[\!]\, os \end{array}$$

where these follow from

- progress-safety-progress on

  $$\begin{array}{ll} v.rc.u \mapsto u.rcv(R.v) & \textbf{in } user \,[\!]\, os \\ u.prec(R.v, D) \wedge u.se \textbf{ unless } u.rcv(R.v) \wedge u.se & \textbf{in } server_u \end{array}$$

  where the safety property follows from conjunction on

  $$\begin{array}{ll} u.prec(R.v, D) \textbf{ unless } u.rcv(R.v) & \textbf{in } server_u \\ u.se \textbf{ unless } u.rcv(D) & \textbf{in } server_u \end{array}$$

- and conjunction on

  $$\begin{array}{ll} u.se \textbf{ unless } u.rcv(D) & \textbf{in } server_u \\ u.rcv(R.v) \wedge (u.se \vee u.clean.v) \textbf{ ensures } u.r.v & \textbf{in } server_u \end{array}$$

- and progress-safety-progress on

$$\begin{array}{ll} u.se \mapsto u.rcv(D) & \textbf{in } user \,[\!]\, os \\ u.r.v \wedge u.f.v \textbf{ unless } u.rcv(D) \wedge u.r.v & \textbf{in } server_u \end{array}$$

  where the safety property was deduced earlier. Receiving the *done* message depends on the client progress and follows from the translation of $u.dine$ which gives

$$u.se \Rightarrow u.pe \vee (u.ph \wedge u.E) \vee u.D$$

  and

$$\begin{array}{ll} u.ph \wedge u.E \textbf{ ensures } u.pe & \textbf{in } client_u \\ u.pe \textbf{ ensures } u.pt \wedge u.D & \textbf{in } client_u \\ u.D \mapsto u.rcv(D) & \textbf{in } user \,[\!]\, os \end{array}$$

5.3.3 Reaching the Forksready State

$$u.mayeat' \mapsto u.forksready \vee \neg(u.h' \wedge u.mayeat') \quad \textbf{in } user \,[\!]\, os$$

can be derived by completion given

$$\begin{array}{rcl} u.mayeat.v & \equiv & u.forkready.v \vee v.fc.u \\ u.forkready.v & \equiv & u.f.v \wedge (u.clean.v \wedge v.r.u) \end{array}$$

and properties

$$\begin{array}{ll} u.mayeat.v \mapsto u.forkready.v \vee \neg(u.h' \wedge u.mayeat') & \textbf{in } user \,[\!]\, os \\ u.forkready.v \textbf{ unless } \neg(u.h' \wedge u.mayeat') & \textbf{in } user \,[\!]\, os \end{array}$$

where

- the progress follows from $v.fc.u \mapsto u.forkready.v$ **in** $user \,[\!]\, os$ which again follows from cancellation on

$$\begin{array}{ll} v.fc.u \mapsto u.rcv(F.v) & \textbf{in } user \,[\!]\, os \\ u.rcv(F.v) \wedge u.miss = 1 \textbf{ ensures} & \\ \quad u.f.v \wedge u.clean.v \wedge u.starteat & \textbf{in } server_u \\ u.rcv(F.v) \wedge u.miss \neq 1 \textbf{ ensures } u.f.v \wedge u.clean.v & \textbf{in } server_u \end{array}$$

- and the safety property follows from simple disjunction and weakening on

$$\begin{array}{ll} u.f.v \wedge u.clean.v \textbf{ unless } \neg u.h' & \textbf{in } server_u \\ u.f.v \wedge v.r.u \textbf{ unless } \neg u.mayeat.v & \textbf{in } server_u \end{array}$$

where the first follows from

$$\begin{array}{ll} u.f.v \wedge u.clean.v \ \mathbf{unless}\ u.se \wedge u.E & \mathbf{in}\ server_u \\ \mathbf{invariant}\quad u.E \Rightarrow \neg u.D & \mathbf{in}\ user \,[\!]\, os \end{array}$$

and given **invariant** $v.fc.u \Rightarrow \neg u.r.v$ **in** $server_u$, the second follows from weakening the conjunction of

$$\begin{array}{ll} u.f.v \ \mathbf{unless}\ u.fc.v & \mathbf{in}\ server_u \\ u.r.v \ \mathbf{unless}\ u.rc.v & \mathbf{in}\ server_u \end{array}$$

where each of these again follows by cancellation on

$$\begin{array}{ll} u.f.v \ \mathbf{unless}\ u.passfork.v & \mathbf{in}\ server_u \\ u.passfork.v \ \mathbf{unless}\ u.fc.v & \mathbf{in}\ server_u \end{array}$$

and

$$\begin{array}{ll} u.r.v \ \mathbf{unless}\ u.passrequest.v & \mathbf{in}\ server_u \\ u.passrequest.v \ \mathbf{unless}\ u.rc.v & \mathbf{in}\ server_u \end{array}$$

5.4 Derived Channel Properties

The refinement of the properties regarding receiving messages left us with some unresolved properties concerning messages in the channel. We now attend to these.

$$\begin{array}{ll} v.rc.u \ \mathbf{unless}\ u.rcv(R.v) & \mathbf{in}\ server_u \\ u.D \ \mathbf{unless}\ u.rcv(D) & \mathbf{in}\ server_u \\ u.prec(R.v, D) \ \mathbf{unless}\ u.rcv(R.v) & \mathbf{in}\ server_u \end{array}$$

The first two follows from $server_u$ being the only receiver on its input channel. The third follows from the FIFO nature of the channel.

$$\begin{array}{ll} v.fc.u \mapsto u.rcv(F.v) & \mathbf{in}\ user \,[\!]\, os \\ v.rc.u \mapsto u.rcv(R.v) & \mathbf{in}\ user \,[\!]\, os \\ u.W \mapsto u.rcv(W) & \mathbf{in}\ user \,[\!]\, os \\ u.D \mapsto u.rcv(D) & \mathbf{in}\ user \,[\!]\, os \end{array}$$

These follow from the channel progress, given that the server eventually becomes ready to receive input, and when there is a message, the server does indeed receive it.

invariant $u.starteat \vee u.passreqs \vee u.readyrecv$	**in** $server_u$
$u.starteat$ **ensures** $u.readyrecv$	**in** $server_u$
$u.passreqs$ **ensures** $u.readyrecv$	**in** $server_u$
$u.rcv(M) \mapsto u.recvd(M)$	**in** $user \| os$

5.5 Properties after Progress Refinement

A few extra safety properties were introduced during the progress refinement:

invariant $u.f.v \wedge u.r.v \Rightarrow u.se$	**in** $server_u$
invariant $u.D \Rightarrow u.se$	**in** $user \| os$
invariant $u.sh \wedge u.cleanornoreq \Rightarrow u.starteat$	**in** $server_u$
invariant $u.starteat \vee u.passreqs \vee u.readyrecv$	**in** $server_u$
$\neg u.f.v$ **unless** $u.rcv(F.v)$	**in** $server_u$
$u.f.v \wedge \neg u.clean.v$ **unless** $u.passfork.v$	**in** $server_u$

Below we list the progress properties derived as a result of the previous refinement steps. To ensure that messages are indeed removed from the channel when received, all ensures properties involving message receiving are strengthened by adding the appropriate $u.recvd(M)$ conjunct. Since consequence weakening allows us to deduce the simpler properties this does not violate the correctness of the refinement. (Also all properties are properties of $server_u$, so for ease of notation the **in** $server_u$ part has been left out in this subsection.)

$u.rcv(D) \wedge u.r.v$ **ensures** $u.recvd(D) \wedge u.st \wedge u.fc.v$

$u.rcv(D) \wedge \neg u.r.v$ **ensures** $u.recvd(D) \wedge u.st \wedge u.f.v \wedge \neg u.clean.v$

$u.rcv(W) \wedge u.miss = 0$ **ensures** $u.recvd(W) \wedge u.sh \wedge u.starteat$

$u.rcv(W) \wedge u.miss \neq 0$ **ensures** $u.recvd(W) \wedge u.sh \wedge u.passreqs$

$u.rcv(F.v) \wedge u.miss = 1$ **ensures**

$\quad u.recvd(F.v) \wedge u.f.v \wedge u.clean.v \wedge u.starteat$

$u.rcv(F.v) \wedge u.miss \neq 1$ **ensures** $u.recvd(F.v) \wedge u.f.v \wedge u.clean.v$

$u.rcv(R.v) \wedge (u.se \vee u.clean.v)$ **ensures** $u.recvd(R.v) \wedge u.r.v$

$u.rcv(R.v) \wedge u.st$ **ensures** $u.recvd(R.v) \wedge u.fc.v$

$u.rcv(R.v) \wedge u.sh \wedge \neg u.clean.v$ **ensures** $u.recvd(R.v) \wedge u.fc.v$

$u.starteat$ **ensures** $u.readyrecv \wedge u.se \wedge u.E$

$u.passrequest.v$ **ensures** $u.rc.v$

$u.passreqs$ **ensures** $u.readyrecv$

From these properties we derive a distributed UNITY program in the next section.

6 Implementations

6.1 A Distributed UNITY Implementation

Figure 6 shows the UNITY program for component $server_u$ of the distributed os program. We have assumed some globally declared variables:

E - is the representation of the graph.

N - is the number of philosophers in the graph.

ch - is the array of server channels.

tc - is the channel to $client_u$.

We will not prove that the progress properties are fulfilled, however, we will prove all the invariants.

6.2 Correctness of UNITY Implementation

Several of the invariants are not true invariants in the sense that they cannot be proved independently. They may only be derived from stronger invariants using the substitution theorem.[1]

Instead of proving each invariant separately we therefore combine them in three stronger invariants which are then proved. (Strictly speaking we still apply the substitution theorem in the proof of the third and most complex invariant.)

$$\begin{array}{lll}
\textbf{invariant} & \neg E[u,v] \vee ExactlyOneOf(u.r.v, u.rc.v, v.r.u, v.rc.u) & \textbf{in } user \,[\!]\, os \\
\textbf{invariant} & E[u,v] \vee (\ u.f.v \wedge \neg u.clean.v \wedge \neg u.r.v \wedge & \\
& \qquad\qquad v.f.u \wedge \neg v.clean.u \wedge \neg v.r.u \wedge & \\
& \qquad\qquad NoneOf(u.fc.v, u.rc.v, v.fc.u, v.rc.u)) & \textbf{in } user \,[\!]\, os \\
\textbf{invariant} & (u.T \vee u.H \vee u.E) \wedge (v.T \vee v.H \vee v.E) \wedge & \\
& a \wedge u.b \wedge v.b \wedge u.c \wedge v.c & \textbf{in } user \,[\!]\, os
\end{array}$$

[1]Please refer to [Bin92] for a more detailed discussion of the notion of invariant in UNITY versus the general definition of the term.

```
Program u.server(N,E,ch,tc)
declare
    f,r,clean : array [0..N-1] of Boolean
    act : {ar, as, ap }
    sdine : {t, h, e}
initially
    ∀ v : ¬E[u,v] ∨ u ≤ v : f[v] ∧ ¬clean[v] ∧ ¬r[v]
    ∀ v : E[u,v] ∧ u > v : ¬f[v] ∧ clean[v] ∧ r[v]
    act = ar ∧ sdine = t
always
    miss : number of v for which f[v] is false
    rcv(M) : readyrecv ∧ hd(ch[u]) = M
    st,se,sh : sdine = t,e,h
    passreqs,starteat,readyrecv : act = ap,as,ar
assign
    s0: ⟨rcv(D)  →  ch[u] := tl(ch[u]) ‖ sdine := t ‖
                    (‖ v: r[v] : ch[v],f[v] := ch[v]⌒(F.u),false) ⟩
[]
    s1: ⟨rcv(W) ∧ miss=0  →  ch[u] := tl(ch[u]) ‖ act,sdine := as,h ⟩
[]
    s2: ⟨rcv(W) ∧ ¬miss=0  →  ch[u] := tl(ch[u]) ‖ act,sdine := ap,h ⟩
[]
    s3: ⟨rcv(F.v) ∧ miss=1  →  ch[u] := tl(ch[u]) ‖ act,f[v],clean[v] := as,true,true ⟩
[]
    s4: ⟨rcv(F.v) ∧ ¬miss=1  →  ch[u] := tl(ch[u]) ‖ f[v],clean[v] := true,true ⟩
[]
    s5: ⟨rcv(R.v) ∧ (se ∨ clean[v])  →  ch[u] := tl(ch[u]) ‖ r[v] := true ⟩
[]
    s6: ⟨rcv(R.v) ∧ st  →  ch[u] := tl(ch[u]) ‖
                           ch[v],f[v],r[v] := ch[v]⌒(F.u),false,true ⟩
[]
    s7: ⟨rcv(R.v) ∧ sh ∧ ¬clean[v]  →  ch[u] := tl(ch[u]) ‖
                                       act,ch[v],f[v],r[v] := ap,ch[v]⌒(F.u),false,true ⟩
[]
    s8: ⟨passreqs  →  (‖ v: ¬f[v] ∧ r[v] : ch[v],r[v] := ch[v]⌒(R.u),false) ‖ act := ar ⟩
[]
    s9: ⟨starteat  →  (‖ v:: clean[v] := false) ‖ act,tc,sdine := ar,tc⌒(eat),e ⟩
end
```

Figure 6: UNITY Program for $server_u$ Component of os

where

$$\begin{array}{rcl}
a & \equiv & \neg E[u,v] \vee ExactlyOneOf(u.f.v, u.fc.v, v.f.u, v.fc.u) \\
u.b & \equiv & (\forall v :: \neg u.r.v \vee \neg v.fc.u) \\
u.c & \equiv & (\forall v :: \neg u.rc.v \vee v.f.u \vee (u.fc.v \wedge v.prec(F.u, R.u))) \\
u.T & \equiv & u.st \wedge (u.rW = u.sE = u.rD) \wedge u.readyrecv \wedge \\
 & & (\forall v :: (u.f.v \ \mathbf{xor}\ u.r.v) \wedge \neg u.clean.v) \\
u.H & \equiv & u.sh \wedge (u.rW = u.sE + 1 = u.rD + 1) \wedge u.starteat = (\forall v :: u.f.v) \wedge \\
 & & (u.passreqs \vee (\forall v :: \neg u.r.v \vee (u.clean.v \wedge u.f.v))) \\
u.E & \equiv & u.se \wedge (u.rW = u.sE = u.rD + 1) \wedge u.readyrecv \wedge \\
 & & (\forall v :: u.f.v \wedge \neg u.clean.v)
\end{array}$$

Correctness and Locality All of the above three properties are stable in *user* since none of the variables can be accessed by the clients. The invariance then follows if the properties are invariant in *os*. Combining the initializations of the servers show that the properties are fulfilled initially. It remains to be proven that they are stable.

1. Since the first property is symmetric in u and v it suffices to prove its stability for $server_u$, the composition principle then allow us to deduce the stability in *os*.
2. Again it suffices to look at $server_u$, and since $E[u,v]$ is constant we may even leave that term out.
3. For the last and most complex property we split the proof obligations. Since $(v.T \vee v.H \vee v.E)$ is local to $server_v$ this term is trivially stable in $server_u$, so we just prove the stability of the remaining terms.

Proof for

$$\textbf{stable} \quad \neg E[u,v] \vee ExactlyOneOf(u.r.v, u.rc.v, v.r.u, v.rc.u) \quad \textbf{in } server_u$$

(s0-s9 refers to the statements in the server program)

s0-s4: no effect.$\surd$

s5-s7: $v.rc.u$ becomes false, but $u.r.v$ true. $\surd$

s8: $u.rc.v$ becomes true, but $u.r.v$ false. $\surd$

s9: no effect.$\surd$

Proof for

$$\textbf{stable} \quad u.f.v \wedge \neg u.clean.v \wedge \neg u.r.v \wedge v.f.u \wedge \neg v.clean.u \wedge \neg v.r.u \wedge NoneOf(u.fc.v, u.rc.v, v.fc.u, v.rc.u) \ \textbf{in} \ server_u$$

s0: $\neg u.r.v$, so no effect.$\surd$

s1-s2: no effect.$\surd$

s3-s4: $\neg v.fc.u$ implies the statement is not enabled.$\surd$

s5-s7: $\neg v.rc.u$ implies the statement is not enabled.$\surd$

s8: $u.f.v$, so no effect.$\surd$

s9: $\neg u.clean.v$ already, so no effect.$\surd$

Proof for

$$\textbf{stable} \quad (u.T \vee u.H \vee u.E) \wedge a \wedge u.b \wedge v.b \wedge u.c \wedge v.c \ \textbf{in} \ server_u$$

For brevity we refer to $u.T$, $u.H$, and $u.E$ as simply T, H and E in the following. $u.b \wedge v.b$ and $u.c \wedge v.c$ are referred to as b and c respectively.

s0: is only enabled if E holds (subproof 1). $\{E\}$s0$\{T\}$, $\{E \wedge a\}$s0$\{a\}$, $u.b$ is not affected, and $v.b$ is preserved, since $u.fc.v$ may become true but only if $u.r.v$ holds implying $\neg v.r.u$ due to invariant (29). Finally $\{E \wedge c\}$s0$\{c\}$. $\surd$

s1,s2: is only enabled if T holds (subproof 2). $\{T\}$s1$\{H\}$, since $miss = 0$ and T implies $\forall v :: \neg u.r.v \wedge u.f.v$. $\{T\}$s2$\{H\}$, since T implies $\forall v :: \neg u.clean.v$. Finally a, b, and c are not affected. $\surd$

s3,s4: is only enabled if H holds (subproof 3). $\{H \wedge a\}$s3$\{H\}$, since $u.f.v$ must be the missing fork. $\{H\}$s4$\{H\}$. For a, $v.fc.u$ becomes false, but $u.f.v$ true. b is not affected. $u.c$ is not affected, and $v.c$ holds since $u.f.v$ becomes true. $\surd$

s5: is only enabled if $E \vee (H \wedge u.clean.v)$ holds. $\{H \wedge u.clean.v\}$s5$\{H\}$, and also $\{E\}$s5$\{E\}$. a is not affected. Also $\{b \wedge v.c\}$s5$\{b\}$, since $v.c \wedge u.rcv(R.v) \Rightarrow \neg v.fc.u$. Finally the request in the channel is removed so $\neg v.rc.u$ is a postcondition, i.e. $\{v.c\}$s5$\{v.c\}$ and for $u.c$, $E \vee u.clean.v \Rightarrow u.f.v$ which is not affected. $\surd$

s6: is only enabled if T. $\{T \wedge a\}$s6$\{T \wedge a\}$ is easily seen. Given invariant (29) $T \wedge a \wedge u.rcv(R.v) \Rightarrow \neg u.rc.v \wedge u.f.v \wedge \neg v.fc.u$, so $u.b$ holds even though $u.r.v$ becomes true, since $v.fc.u$ is not affected. Finally $v.b$ is also preserved, since postcondition $u.r.v$ implies $\neg v.r.u$ according to invariant (29), $v.c$ is preserved, since $v.rc.u$ becomes false, and $u.c$ holds since $\neg u.rc.v \wedge u.fc.v$ is part of the postcondition. $\surd$

s7: is only enabled if H. $\{H\}$s7$\{H\}$, and again given invariant (29) $u.c \wedge u.rcv(R.v)$ implies $\neg v.fc.u \wedge u.f.v \wedge \neg u.rc.v$, so $\{u.c\}$s7$\{a \wedge u.c \wedge v.c \wedge b\}$. $\surd$

s8: is only enabled if H. $\{H\}$s8$\{H\}$, a is not affected, $\neg u.r.v$ as postcondition implies $u.b$, and $v.b$ is not affected. Finally given invariant (29) precondition $u.r.v$ implies $\neg v.rc.u$, so $v.c$ is preserved. For $u.c$ precondition $u.r.v \wedge \neg u.f.v$ with $u.c$ and b gives $(v.prec(F.u, R.u) \wedge u.fc.v) \vee v.f.u)$ which still holds after execution of s8. $\surd$

s9: is only enabled if H. $\{H\}$s9$\{E\}$ is immediate, and a, b and c are not affected. $\surd$

Subproofs:

1.

$$
\begin{array}{ll}
 & u.rcv(D) \\
\Rightarrow & \qquad \{ \text{ definition of } sX \text{ and } rX \ \} \\
 & u.sD > u.rD \\
\Rightarrow & \qquad \{ \text{ client invariant (14) } \} \\
 & u.rE > u.rD \\
\Rightarrow & \qquad \{ \text{ channel invariant } sX \geq rX \ \} \\
 & u.sE > u.rD \\
\Rightarrow & \qquad \{ \text{ precondition } T \vee H \vee E \ \} \\
 & E
\end{array}
$$

2.

$$
\begin{array}{ll}
 & u.rcv(W) \\
\Rightarrow & \qquad \{ \text{ definition of } sX, rX \text{ and} \\
 & \qquad\ \ \text{client invariant (15) } u.D \wedge u.W \Rightarrow u.prec(D, W) \ \ \} \\
 & u.sW > u.rW \wedge u.rD = u.sD \\
\Rightarrow & \qquad \{ \text{ client invariant (14) } \} \\
 & u.sW > u.rW \geq u.rD = u.sD \geq u.sW - 1 \\
\Rightarrow & \qquad \{ \text{ math } \} \\
 & u.rW = u.rD \\
\Rightarrow & \qquad \{ \text{ precondition } T \vee H \vee E \ \} \\
 & T
\end{array}
$$

3. $u.rcv(F.v)$

$\Rightarrow$ { definition of $u.rcv(F.v)$ and $v.fc.u$ }

$v.fc.u$

$\Rightarrow$ { preconditions a and c }

$\neg u.f.v \wedge \neg u.r.v$

$\Rightarrow$ { precondition b and $(T \vee H \vee E)$ }

H

6.3 CC++ Implementation

The UNITY program may almost literally be translated to CC++. The main difference is that the parallel assignments in the actions are replaced by sequential assignments. Also *miss* must be updated explicitly.

```
#include "Channel.h"
#include "Diners.h"
#include "Server.h"

void client(int u, Channel<Msg>& ServerCh, Channel_void& ToClient)
{ for (;;)
  { think(u); ServerCh.nonblockingSend({want,u});
    ToClient.blockingReceive();
    eat(u); ServerCh.nonblockingSend({done,u});
} }

int main()
{ enum   MsgType { want, done, fork, request }
  enum   State   { t, h, e }
  struct Msg     { MsgType type, int u }
  Channel<Msg> ServerCh[N];
  Channel_void ToClient[N];
  for (int v=0; v<N; v++)
  { spawn client(v, ServerCh[v], ToClient[v]);
    spawn server(v, ServerCh, ToClient[v]);
  }
}
```

Figure 7: Distributed Main and Client Program in CC++.

Including the program text between /* and */ gives an implementation with the same grain of atomicity as the UNITY program. However the finer grain which is achieved without the comment parts is still correct. We will not prove this claim, however. For completeness we have enclosed a client definition and a main program that sets up the system. Assignments to the *cdine* variable are not included, since they do not affect the communication. Note: some obvious optimizations for a C/C++ programmer has been left out to obtain a program more similar to the UNITY program.

Figure 7 illustrates a possible main and client program. The main program resembles that of our simple example page 8 only here we declare 2N channels and spawn 2N processes (or parallel threads).

```
void Server(int u, Channel<Msg>[N]& ch, Channel_void& tc)
{ enum      Action  { ar, as, ap }
  Boolean   f[N],clean[N],r[N];
  State     sdine = t;
  int       miss = 0;
  Action    act = ar;
  const     struct  Msg     forkMsg = {fork,u};
  const     struct  Msg     reqMsg  = {request,u};

  initialize();
  for (;;)
  switch (act)
  { case ar:  /*ch[u].awaitmessage(); */ receiveMsg(); break;
    case as:  starteat();                             break;
    case ap:  passrequest();                          break;
} }

void Server::initialize()
{ for (int v=0; v<N; v++)
  { if (!E[u,v] || (u <= v)) { f[v]=true; clean[v]=false; r[v]=false; }
    else { f[v]=false; clean[v]=true; r[v]=true; miss++; }
} }
```

Figure 8: Initialization and Top-level for $server_u$ in CC++.

The fork and request messages consist of message type as well as sender identification, so the server input channels hold structured messages. The client channels are declared as void channels. Since only one type of message (Eat) is transferred a simple signal suffices.

The client performs an infinite loop. Each time think terminates (if ever) a want message is sent to the server. The client then waits for the void message on ToClient before the eat function begins. Termination of eat is signalled to the server by sending a done message.

The Diners.h file is assumed to contain proper definitions of the global data structures N, E[N,N], the client functions think and eat and the enumeration data type Boolean.

The top-level of the server function in CC++ contains the declarations and initializations from the *u.server* UNITY program (see figure 6 page 30). The initial state of the composed UNITY program has no counterpart in the execution of the CC++ program, since servers and clients do not synchronize after initialization of their variables. For each component, however the loop `for(;;)` ... in the CC++ program corresponds to the assign section of the UNITY program.

The switch on act at the top-level corresponds to the first conjunct of the UNITY actions: the first case corresponds to actions s0-s7, the second to s8, and the third to s9. If we want to mimic the atomicity level of the UNITY program, we cannot use a blocking receive for actions s0-s7. The blocking receive may suspend so we cannot guarantee its termination and it is hence not a legal statement in an atomic function. Instead we first wait for an available message and then receive it using a non-blocking atomic receive function. Since each channel has only one receiver, the available message will not disappear unless the server receives it.

```
/*atomic*/ void Server::receiveMsg()
{ struct Msg * sync msg;
  ch[u]./*non*/blockingReceive(&msg);
  switch (msg->type)
  { case done:        handleDone();           break;
    case want:        handleWant();           break;
    case fork:        handleFork(msg->v);     break;
    case request:      handleRequest(msg->v); break;
} }

/*atomic*/ void Server::starteat()
{ tc.nonblockingSend(); sdine=eating; act=ar;
  for (int v=0; v<N; v++) clean[v]=false;
}

/*atomic*/ void Server::passrequests()
{ act=ar; for (int v=0; v<N; v++) if (!f[v])
  { ch[v].nonblockingSend(reqMsg); r[v]=false; }
}
```

Figure 9: Receive Message, Start Eat, and Pass Request for $server_u$ in CC++.

The switch on `msg->type` in `receiveMsg` corresponds to the hd(ch[u])=M test in the UNITY program. The first case corresponds to action s0, the second to s1-s2, the third to s3-s4, and the last to s5-s7.

The parallel and multiple assignments in s8 and s9 are done sequentially in `startEat` and `passRequests`, but since the assignments are independent the semantics are unaffected.

```
void Server::handleDone()
{ sdine=t;
  for (int v=0; v<N; v++) if (r[v])
  { ch[v].nonblockingSend(forkMsg); f[v]=false; miss++; }
}

void Server::handleWant()
{ sdine=h; if (miss==0) act=as; else act=ap; }

void Server::handleFork(const int v)
{ f[v]=true; clean[v]=true; miss--; if (miss==0) act=as; }

void Server::handleRequest(const int v)
{ r[v]=true; if (sdine==h && !clean[v]) act=ap;
  if (sdine!=e && !clean[v])
  { ch[v].nonblockingSend(forkMsg); f[v]=false; miss++; }
}
```

Figure 10: Message Handling Functions for $server_u$ in CC++.

For actions s0-s7 we have used embedded tests to handle the actions in groups: s0, s1-s2, s3-4, s5-s7. Again the parallel and multiple assignments are done in sequence. Though the UNITY assignments are independent, we must pay attention in `handleFork`; updating *miss* is now part of the CC++ program and when we update *miss* before the test we must adjust the test accordingly.

7 Conclusions and Future Work

We have shown how an executable distributed CC++ program may be derived from a shared memory UNITY specification. Quite a few details of the transformations have been left out, though. Many of the proofs are only sketched. The complete proofs are too comprehensive. The channel description is still too vague for a mechanical verification. Finally the ad hoc nature of the UNITY to CC++ transformation is insufficient for a truly formal proof.

So what are the conclusions? First of all, that a systematic and quite formal derivation from specification to implementation is possible. Second, that it is very time consuming. The latter is probably no surprise to anyone who has worked with formal proofs, however, in this case there is some hope; All the proofs in this report were done by hand, but experiments with an earlier example—mutual exclusion in a token-ring [Bin92]—show that many of the proofs could as well be done mechanically. Ongoing work on mechanically verifying the properties in the afore mentioned example using HOL-UNITY [And92] shows that tactics can be written to prove the properties automatically. Knowledge of the detailed steps of the proofs are not important to complete the mechanical proofs with these tactics. Rather the locality and composition of properties seem to be the hurdle. These aspects are often only treated rather informal in the proof done by hand, but are crucial to the mechanical proof.

The missing details of the proofs in this report is thus not important to the mechanical verification. We have tried to be somewhat more precise regarding the locality of variables and properties, but this still needs improvement.

At this point we can mechanically verify 1) that a given UNITY program fulfills a set of properties, and 2) the correctness of property refinements. As a further step towards a formal verification, we will focus on defining a formal semantics in UNITY for a subset of CC++. This would provide the missing link between CC++ and UNITY.

If the CC++ semantics are formalized in HOL-UNITY, we have a complete mechanical verification path: verify that a given CC++ program corresponds to a UNITY program that fulfills a set of properties, which again corresponds to the specification via a series of refinement step.

A complete mechanical derivation tool, i.e. from specification to implementation, would likewise require a UNITY property to program translator and a UNITY program to CC++ program translator.

Acknowledgment

The further refinement of the final *os* specification in [CM88] as opposed to a refinement from scratch was inspired by Singh's work on program refinement [Sin92].

Mani Chandy inspired the functional programming idea behind the CC++ program, and I would like to thank him for his very valuable support during my work. Also thanks to Rustan Leino and Peter Hofstee for a good push, when I was stuck on an invariant proof. Last but not least thanks to everyone in Mani Chandy's group at Caltech for many good discussions and their extensive comments to an earlier version of this report.

References

[And92] Flemming Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Dept. of Computer Science, Technical University of Denmark, March 1992.

[Bin92] Ulla Binau. Mutual exclusion in a token ring in CC++: Program and proof. Technical Report Caltech-CS-TR-92-11, California Institute of Technology, May 1992.

[Bin93] Ulla Binau. Distributed diners in CC++: The "complete" proof. Technical report, California Institute of Technology, 1993. To be pulished.

[CK92] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. Technical Report Caltech-CS-TR-92-01, California Institute of Technology, 1992.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Langauge*. Prentice Hall, 1988.

[Sin92] Ambuj K. Singh. Program refinement in fair transition systems. Technical report, Dept. of Computer Science, UCSB, 1992. An earlier version appears in LNCS 506, p.128-147.

[Siv93] Paul Sivilotti. A verified integration of imperative parallel programming paradigms in an object-oriented language. Technical Report Caltech-CS-TR-93-21, California Institute of Technology, 1993.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, AT&T Bell Laboratories, 2nd edition, 1991.