Mutual Exclusion in a Token Ring in CC++: Program and Proof

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

Caltech-CS-TR-92-11
May 18, 1992

Abstract

This report describes a first attempt at using UNITY to verify reactive Compositional C++ (CC++) programs. We propose a distributed solution to the mutual exclusion problem using partially synchronous communication channels. The solution is described as a CC++ program, from which a small set of "basic" properties is derived. Using UNITY, we proof mutual exclusion and progress of the solution based on the set of properties derived from the code.

1 Introduction

CC++ is C++ [11] extended with a small set of constructs that allow design of parallel programs. Assuming that the parallel execution of CC++ statements is fair, we may apply the UNITY proof theory to our CC++ programs.

This report describes a first attempt at using UNITY to assist verification of safety and progress properties for reactive CC++ programs. The mutual

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

exclusion problem (originally posed by Dijkstra [5]) was chosen as a first example due to its simplicity (assuming availability of atomic operations at least at the level of "test-and-set").

The problem specification was inspired by the description of the related dining philosophers problem in [4]. The solution given in this report—a variation of LeLann's token-passing solution [7]—is due to Mani Chandy, who has also contributed extensively to the proof.

The original idea was to stepwise refine a UNITY program, and finally transform this to a CC++ program. However, since the goal program is highly sequential, a sufficiently fine-grained UNITY description turned out to be very hard to read compared to the CC++ program itself. Because of that, we decided to leave out the UNITY programs entirely, and merely present the CC++ program.

The program uses communication functions from a communication library [3]. Avoiding discussions of implementation details, we give a short description and list the properties of the functions used in this context.

The UNITY proof of the mutual exclusion program is based on a small set of simple properties derived directly from the main program functions (using the properties of the communication functions). All safety and progress properties depend merely on local and/or stable predicates. This implies that all properties of subcomponents of the system are also properties of the whole system. In fact the proof is compositional.

The report is organized as follows: We describe the problem and the solution idea, define our requirements to the communication functions, present the complete CC++ solution, and give annotated versions of the main functions of the program. From the annotated functions we then derive the basic properties used in the following proofs of, first, safety properties, and second, progress properties. We round up with a few concluding remarks and an appendix describing the proof style and listing the UNITY proof rules applied in the progress proofs.

We have chosen neither to give an introduction to UNITY nor to CC++, since both are described elsewhere (see especially [4] and [1, 2]).

2 Problem Specification

Informal Description We are given N clients indexed i, where $0 \leq i<N$. A client is in one of three states (1) thinking, (2) hungry, or (3) eating. The only transitions between these states are (a) thinking to hungry, (b) hungry to eating, and (c) eating to thinking. The client executes a given "think" procedure in the thinking state, and a given "eat" procedure in the eating state. In the hungry state it merely waits to enter its eating state. A client can remain thinking forever, but it can only remain eating for a *finite* period of time.

The problem is to design a control system that guarantees

- Safety: No clients eat simultaneously.
- Progress: Every hungry client eats eventually.

We may assume an interface between each client and the control system, so that all transitions of a client are visible to the control system, and transition (b) is made only after receiving permission from the control system. (Transitions (a) and (c) are made by the client without necessarily waiting for communication from the control system.)

Formal Specification Using UNITY the requirements are formally expressed:

- Safety: $< \forall i, j : i \neq j : \neg(eating_i \wedge eating_j) >$
- Progress: $< \forall i :: hungry_i \mapsto eating_i >$

In the following we will assume universal quantification of any free occurrences of i, j or k.

3 Solution Idea

Our solution idea is to have a single token circulating in a ring of servers. Each client then requests and receives the token from one server in the ring and sends it to the next server in the ring. The client may never want to enter the critical section, so if no request from the client is pending, we require each server to pass the token directly to the next server in the ring.

The uniqueness of the token will guarantee mutual exclusion, if a client must hold the token to be in its critical section.

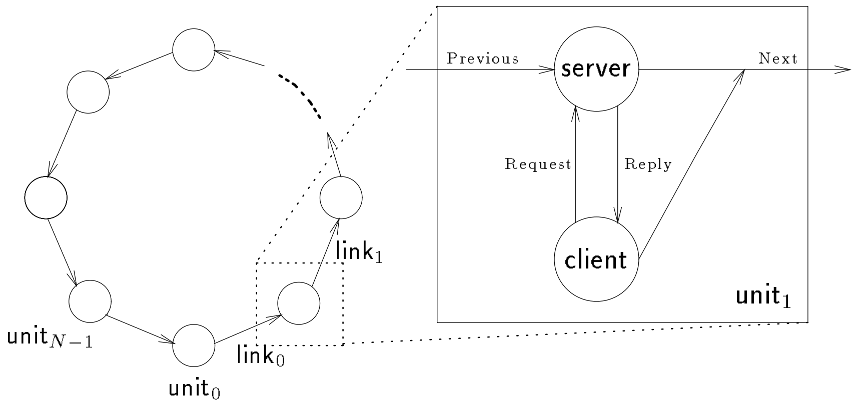

Figure 1: A mutual exclusion system with N units — and a closer look at one of the units

4 Channel Properties

The program uses channels with infinite buffer capacity for synchronization (no actual message is passed, so the channels correspond to semaphores). We use three types of communication functions:

- `send` (nonblocking atomic) corresponds to the traditional V operation.
- `receive` (blocking) corresponds to the traditional P operation.
- `probeReceive` (nonblocking atomic) corresponds to a conditional P operation; if the channel buffer is empty, the call returns false, otherwise the first signal in the channel buffer is consumed, and the call returns true.

Similar to Martin's axiomatic definition of synchronization primitives [8] we define for any channel:

- cS (non-negative, initially zero, cannot decrease) denotes the number of *completed* sends.
- cR (non-negative, initially zero, cannot decrease) denotes the number of *completed* receives.
- qR (non-negative, initially zero) denotes the number of *suspended* receives.
- kR (non-negative, constant) denotes the number of receives that can be completed prior to the first send (i.e., the number of signals in the channel buffer initially).

A completed `probeReceive` call that returned true counts as a completed receive. For programs with more than one channel, the variables are distinguished by postfixing the channel name.

We characterize the channel by its properties:

- Boundedness: **invariant** $cS + kR \geq cR$
- Maximal Progress: **invariant** $(qR = 0) \vee (cS + kR = cR)$
- Fairness: receives are completed on a first-suspended-first-completed basis.
- Grain of Atomicity: at most one receive or one send can be completed in each step.

Notice that the fairness property for the CC++ channel is stronger than the generally used requirement: receives are completed within a finite time. Also we have added a grain of atomicity property.

In general CC++ channels[1] may have any number of senders and receivers. In this context, however, all channels have a single receiver, one channel has two senders, all others, a single sender (compare with figure 1 or the following program).

[1][3] describes the channel objects and their functions in detail. For simplicity (and to increase readability) we use `receive` and `send` instead of the longer names `blockingReceive` and `nonblockingSend`.

5 CC++ Program and Ghost Variables

The following is our proposed distributed CC++ program for the mutual exclusion problem for N clients:

Mutual Exclusion

```
void main()
{ Channel<void> Link[N]{1};    // N links, link 0 holds the token,
                               // link 1 to N-1 all empty
  Channel<void> Request[N];    // N empty request channels
  Channel<void> Reply[N];      // N empty reply channels
  parfor (i = 0; i < N; i++)  // run the N units in parallel
  { par                        // run client and server in parallel
    { client(Request[i], Reply[i], Link[(i+1)%N]);
      server(Link[i], Request[i], Reply[i], Link[(i+1)%N]);
    }
  }
}

void client(Channel<void> Request, Reply, Next)
{ for (;;)
  { think();               // finished thinking, send request
    Request.send();        // request sent, wait for permission
    Reply.receive();       // permission granted, start eating
    eat();                 // finished eating, pass token
    Next.send();           // token passed, return to thinking
  }
}

void server(Channel<void> Previous, Request, Reply, Next)
{ for (;;)
  { Previous.receive();          // token received, check request
    if (Request.probeReceive()) // if requested, grant permission
      Reply.send();              // permission granted
    else                         // else pass token
      Next.send();               // token passed
  }
}
```

Any parameters of `think` and `eat`—the given procedures of the clients—are left out for simplicity.

For each server and client we introduce a set of local ghost variables:

- local ghost variables of client

  cpc program counter.

  sRequest number of sends completed on channel Request (by the client).

  rReply number of receives completed on channel Receive (by the client).

  sClientNext number of sends completed on channel Next by the client.

- local ghost variables of server

  spc program counter.

  rPrevious number of receives completed on channel Previous (by the server).

  rRequest number of receives completed on channel Request (by the server).

  sReply number of sends completed on channel Reply (by the server).

  sServerNext number of sends completed on channel Next by the server.

As useful shorthands we define

$$outServer \equiv sReply + sServerNext$$
$$sNext \equiv sClientNext + sServerNext$$

Since the units are connected in a ring, Next of unit i corresponds to Previous of unit $i \oplus 1$ (and vice versa). Using $\oplus$ and $\ominus$ to denote addition and subtraction modulo the number of units in the ring, we get:

Connections: $Link_i = Previous_i = Next_{i \ominus 1}$

Relating the ghost variables to the variables, cR, cS, and kR defined earlier for each channel, we find:

Initializations: $kR.Reply = kR.Request = 0$
$kR.Link_0 = 1$
$kR.Link_i = 0$, for $i \neq 0$

Invariants: $cR.Link_i = rPrevious_i$ and $cS.Link_i = sNext_{i\ominus 1}$
$cR.Request_i = rRequest_i$ and $cS.Request_i = sRequest_i$
$cR.Reply_i = rReply_i$ and $cS.Reply_i = sReply_i$

We will later refer to $kR.Link_i$ as $iPrevious_i$, and to $sNext_{i\ominus 1}$ as $sPrevious_i$

The channel properties and the above shorthands, connections, initializations and invariants allow us to deduce for $C = Previous$, $Request$, $Reply$, $Next$:

(1) $\textbf{stable} \quad sC \geq k$

(2) $\textbf{invariant} \quad (rC \geq 0) \wedge (sC \geq 0)$

(3) $\textbf{invariant} \quad sRequest \geq rRequest$

(4) $\textbf{invariant} \quad sReply \geq rReply$

(5) $\textbf{invariant} \quad sNext_{i-1} \geq rPrevious_i, \quad i = 1, \ldots, N-1$

(6) $\textbf{invariant} \quad sNext_{N-1} + 1 \geq rPrevious_0$

(7) $(sRequest \geq k) \wedge (rRequest = k-1) \textbf{ unless } (rRequest \geq k)$

In the following annotated versions of the client and server functions we will use /! and !/ to indicate CC++ code that assigns the local ghost variables. This code is assumed to be executed atomically with the termination of the "real" CC++ code that it follows.

6 Compositionality, Invariants and Substitution Axiom

In the following we several times deduce *invariant* q from *invariant* p and $p \Rightarrow q$. This gave rise to a discussion about *invariant* versus *always-true* among those who read an earlier version of this report. Jan van de Snepscheut brought our attention to the problem as well as the references mentioned in this section. Given alone the definitions

- A predicate p is *invariant* in program F, if and only if, p holds initially in F, and every *statement* s in F preserves p.
- A predicate p is *always-true* in program F, if and only if, p holds initially in F, and p holds in any *reachable state* s in F.

we may only deduce *always-true* q from *invariant* p and $p \Rightarrow q$. Invariants are compositional, but always-true properties are not (see e.g. [12]), so this would not be suitable for our purpose.

However, the deduction we apply does not rely solely on the definition of *invariant*; it involves two implicit applications of the UNITY substitution axiom: 1. from $p \Rightarrow q$ we get the theorem $p \equiv p \wedge q$, now applying the substitution axiom to this and the property *invariant* p, we may substitute $p \wedge q$ for p and get *invariant* $p \wedge q$, 2. from *invariant* p we get the theorem $p \equiv true$, now applying the substitution axiom again, this and *invariant* $p \wedge q$ gives *invariant* $true \wedge q$, i.e., *invariant* q.

The first application will always be correct, but in the second step we use the theorem $p \equiv true$. For the proof to be correct when F is composed with another program G, $p \equiv true$ must be a theorem in the composed system, that is, p must also be an invariant of G.

The problems involved in implicit/explicit use of the substitution axiom is mentioned briefly in section 7.2.4 in [4]. Misra gives a more thorough discussion of the substitution axiom in [10].

The first client invariant and server invariant given below can be proved without application of the substitution axiom and involves only local variables. These invariants are hence also invariants of the complete system and we are free to use them in applications of the substitution axiom.

In general, since all the proofs of the client and server properties are based on predicates on local variables and/or guaranteed stable predicates on non-local variables, any of these proofs will also be valid proofs in the complete system.

7 Client Properties

Annotated Client Program

```
void client(Channel<void> Request, Reply, Next)
/! integer cpc=0, sRequest=0, rReply=0, sClientNext=0; !/
{ for (;;)
  { think();           /! cpc=1; !/
    Request.send();    /! cpc=2; sRequest++; !/
    Reply.receive();   /! cpc=3; rReply++; !/
    eat();             /* ignore since finite and local */
    Next.send();       /! cpc=0; sClientNext++; !/
  }
}
```

Let Cx denote $cpc = x$ then we see that $C1 \vee C2$ corresponds to *hungry* in the original specification and that $C3$ (will lead) to *eating*.

Client Invariants From the annotations of the client program, we find[2]

client invariant IC: **invariant** $\quad IC01 \vee IC2 \vee IC3$

where

$IC01 = (C0 \vee C1) \wedge (sRequest = rReply) \wedge (rReply = sClientNext)$

$IC2 = C2 \wedge (sRequest = rReply + 1) \wedge (rReply = sClientNext)$

$IC3 = C3 \wedge (sRequest = rReply) \wedge (rReply = sClientNext + 1)$

Lemma 1: Only passing received tokens

invariant $\quad rReply \geq sClientNext$

Proof:

$\quad (IC01 \vee IC3) \vee IC2$ (client invariant)

$\Rightarrow$ { simplification and weakening }

$\quad (rReply = sClientNext) \vee (rReply = sClientNext + 1)$

$\Rightarrow$ { math and predicate calculus }

[2]To proof this invariant using UNITY we may assume that each statement is guarded by a predicate on the local program counter, e.g. that the statement `think()` is only executed when $cpc = 0$. The guards correspond to the fixed (sequential) execution order of the statements, and cannot be enabled or disabled by any other process.

$rReply \geq sClientNext$

□

LEMMA 2: AT MOST 1 PENDING REQUEST

invariant $sClientNext \geq sRequest - 1$

Proof:

$(IC01 \vee IC2) \vee IC3$ (client invariant)

$\Rightarrow$ { simplification and weakening }

$(sRequest = sClientNext) \vee (sRequest = sClientNext + 1)$

$\Rightarrow$ { math and predicate calculus }

$sClientNext \geq sRequest - 1$

□

Other Client Properties

LEMMA 3: SEND REQUEST WHEN HUNGRY

$C1 \wedge (sRequest = j - 1) \mapsto C2 \wedge (sRequest = j)$

Follows from the client program, and progress condition for `send` since $sRequest$ is local.

□

LEMMA 4: KEEP WAITING UNLESS TOKEN IS RECEIVED

$C2 \wedge (rReply = j - 1)$ **unless** $C3 \wedge (rReply = j)$

Follows from the client program, since $rReply$ is local.

□

Lemma 5: Receive reply if available

$C2 \wedge (sReply \geq j) \wedge (rReply = j-1) \;\mapsto C3 \wedge (rReply = j)$

Follows from the client program, (1) page 8, locality of $rReply$, and progress condition for `receive`.

□

Lemma 6: Pass token after eating

$C3 \wedge (sClientNext = j-1) \;\mapsto C0 \wedge (sClientNext = j)$

Follows from the client program, and progress condition for `send`, since $sClientNext$ is local.

□

8 Server Properties

Annotated Server Program

```
void server(Channel<void> Previous, Request, Reply, Next)
/! integer spc=0, rPrevious=0, rRequest=0, sReply=0, sServerNext=0; !/
{ for (;;)
  { Previous.receive();         /! spc=1; rPrevious++; !/
    if (Request.probeReceive()) /! spc=2; rRequest++; !/
      Reply.send();             /! spc=0; sReply++; !/
    else                        /! spc=3; !/
      Next.send();              /! spc=0; sServerNext++; !/
  }
}
```

Server Invariants As for the client program we derive (Sx denoting $spc = x$):

server invariant: **invariant** $\quad IS0 \vee IS13 \vee IS2$

where

$IS0 : SO \wedge (rPrevious = outServer) \wedge (rRequest = sReply)$

$IS13 : (S1 \vee S3) \wedge (rPrevious = outServer + 1) \wedge (rRequest = sReply)$

$IS2 : S2 \wedge (rPrevious = outServer + 1) \wedge (rRequest = sReply + 1)$

Lemma 7: Only sending received tokens

invariant $rPrevious \geq outServer$

Proof:

$IS0 \vee (IS13 \vee IS2)$ (server invariant)

$\Rightarrow$ { simplification and weakening }

$(rPrevious = outServer) \vee (rPrevious = outServer + 1)$

$\Rightarrow$ { math and predicate calculus }

$rPrevious \geq outServer$

$\square$

Lemma 8: Only granting requested tokens

invariant $rRequest \geq sReply$

Proof:

$IS0 \vee (IS13 \vee IS2)$ (server invariant)

$\Rightarrow$ { simplification and weakening }

$(rRequest = sReply) \vee (rRequest = sReply + 1)$

$\Rightarrow$ { math and predicate calculus }

$rRequest \geq sReply$

$\square$

Other Server Properties

Lemma 9: Tokens passed on cannot be taken back

stable $sServerNext \geq j$

Follows from the server program since $sServerNext$ is never decreased.

$\square$

LEMMA 10: KEEP WAITING UNLESS TOKEN IS RECEIVED

$S0 \wedge (rPrevious = j - 1)$ **unless** $S1 \wedge (rPrevious = j)$

Follows from the server program and locality of $rPrevious$.

$\square$

LEMMA 11: RECEIVE TOKEN

$S0 \wedge (sPrevious + iPrevious \geq j) \wedge (rPrevious = j - 1) \;\mapsto$
$S1 \wedge (rPrevious = j)$

Follows from the server program, (1) page 8, and progress condition for `receive`, since $iPrevious$ is constant, and $rPrevious$ local to the server.

$\square$

LEMMA 12: RECEIVE REQUEST

$S1 \wedge (sRequest \geq j) \wedge (rRequest = j - 1) \;\mapsto S2 \wedge (rRequest = j)$

Follows from the server program, (1) page 8, and progress condition for `probeReceive`, since $rRequest$ is local to the server.

$\square$

LEMMA 13: CHECKING REQUEST PRODUCES NO OUTPUT

$S1 \wedge (outServer = j - 1) \;\mapsto (S2 \vee S3) \wedge (outServer = j - 1)$

Follows from the server program, definition of $outServer$, and progress condition for `probeReceive`, since $outServer$ is local to the server.

$\square$

Lemma 14: Send reply

$S2 \wedge (sReply = j - 1) \mapsto S0 \wedge (sReply = j)$

Follows from the server program, and progress condition for `send`, since $sReply$ is local to the server.

□

Lemma 15: Pass token

$S3 \wedge (sServerNext = j - 1) \mapsto S0 \wedge (sServerNext = j)$

Follows from the server program, and progress condition for `send`, since $sServerNext$ is local to the server.

□

Lemma 16: Sending reply or passing token produces output

$(S2 \vee S3) \wedge (outServer = k - 1) \mapsto (outServer \geq k)$

Proof:

Assertion (1):

$$
\begin{array}{ll}
 & S2 \wedge (outServer = k - 1) \\
\Rightarrow & \{ \text{ definition of } outServer \ \} \\
 & < \exists i, j : i + j = k : S2 \wedge (sReply = j - 1) \wedge (sServerNext = i) > \\
\Rightarrow & \{ \text{ pick } i \text{ and } j \ \} \\
 & S2 \wedge (sReply = j - 1) \wedge (sServerNext \geq i) \\
\mapsto & \{ \text{ S-PSP with lemma 14 and 9 } \} \\
 & S0 \wedge (sReply = j) \wedge (sServerNext \geq i) \\
\Rightarrow & \{ \text{ math and predicate calculus } \} \\
 & sReply + sServerNext \geq i + j \\
= & \{ \ i + j = k, \text{ and definition of } outServer \ \} \\
 & outServer \geq k
\end{array}
$$

Assertion (2): A similar proof gives

$S3 \wedge (outServer = k - 1) \mapsto (outServer \geq k)$

Simple disjunction of (1) and (2) completes the proof.

$\square$

9 Verification of Safety Properties

THEOREM 1: EATING $\Rightarrow$ HOLDS TOKEN

invariant $C3 \Rightarrow (rPrevious > sNext)$

Proof:

$$
\begin{array}{ll}
 & rPrevious \\
\geq & \{ \text{ lemma 7 } \} \\
 & outServer \\
= & \{ \text{ definition } outServer \equiv sServerNext + sReply \ \} \\
 & sServerNext + sReply \\
\geq & \{ \text{ (4) page 8: } sReply \geq rReply \ \} \\
 & sServerNext + rReply \\
> & \{ \ C3 \text{ and client invariant: } C3 \Rightarrow (rReply = sClientNext + 1) \ \} \\
 & sServerNext + sClientNext \\
= & \{ \text{ definition } sNext \equiv sServerNext + sClientNext \ \} \\
 & sNext
\end{array}
$$

$\square$

LEMMA 17

invariant
$sRequest \geq rRequest \geq sReply \geq rReply \geq sClientNext \geq sRequest - 1$

Proof:

$$\begin{array}{ll} sRequest \geq rRequest & ((3) \text{ page } 8) \\ rRequest \geq sReply & (\text{lemma } 8) \\ sReply \geq rReply & ((4) \text{ page } 8) \\ rReply \geq sClientNext & (\text{lemma } 1) \\ sClientNext \geq sRequest - 1 & (\text{lemma } 2) \end{array}$$

$\Rightarrow$ { conjunction, transitivity of $\geq$, and simplification }

$$sRequest \geq rRequest \geq sReply \geq rReply \geq sClientNext \geq sRequest - 1$$

□

Theorem 2: Tokens are not created

invariant $rPrevious \geq sNext$

Proof:

$rPrevious$

$\geq$ { lemma 7 }

$outServer$

$=$ { definition of outServer }

$sServerNext + sReply$

$\geq$ { lemma 17 }

$sServerNext + sClientNext$

$=$ { definition $sNext \equiv sServerNext + sClientNext$ }

$sNext$

□

Theorem 3: At most 1 token in system

invariant $rPrevious_0 \geq sNext_0 \geq \ldots \geq sNext_{N-1} \geq rPrevious_0 - 1$

Proof:

$$\begin{array}{ll} rPrevious_i \geq sNext_i & (\text{theorem } 2) \\ sNext_{i-1} \geq rPrevious_i, i = 1, \ldots, N-1 & ((5) \text{ page } 8) \\ sNext_{N-1} + 1 \geq rPrevious_0 & ((6) \text{ page } 8) \end{array}$$

$\Rightarrow \quad \{$ conjunction and transitivity of $\geq$ $\}$

$rPrevious_0 \geq sNext_0 \geq rPrevious_1 \geq \ldots \geq sNext_{N-1} \geq rPrevious_0 - 1$

$\Box$

THEOREM 4: MUTUAL EXCLUSION

invariant $\quad \neg C3_i \vee \neg C3_j, i \neq j$

Proof: The proof is by contradiction

$C3_i \wedge C3_j$

$\Rightarrow \quad \{$ theorem 1 $\}$

$(rPrevious_i > sNext_i) \wedge (rPrevious_j > sNext_j)$

$\Rightarrow \quad \{$ $i \neq j$ implies violation of theorem 3 $\}$

$false$

$\Box$

We prove an additional safety property needed for the proof of progress.

LEMMA 18: SERVER IS WAITING IF NO TOKENS IN UNIT

invariant $\quad (rPrevious = sNext) \Rightarrow S0$

Proof:

$outServer \geq rPrevious$ (subproof)
$rPrevious \geq outServer$ (lemma 7)

$= \quad \{$ conjunction of above two $\}$

$rPrevious = outServer$

$= \quad \{$ server invariant $\}$

$S0$

Subproof:

$rPrevious$

$$
\begin{aligned}
= \quad & \{ \text{ assumption } \} \\
& sNext \\
= \quad & \{ \text{ definition } sNext \equiv sClientNext + sServerNext \ \} \\
& sClientNext + sServerNext \\
\leq \quad & \{ \text{ lemma 17: } sClientNext \leq sReply \ \} \\
& sReply + sServerNext \\
= \quad & \{ \text{ definition } outServer \equiv sReply + sServerNext \ \} \\
& outServer
\end{aligned}
$$

□

10 Verification of Progress Properties

In the following proofs we will refer several times to a *simple cancellation* rule. Please refer to the appendix for a proof of this rule.

LEMMA 19: SERVER SENDS OUT ITS TOKEN EVENTUALLY

$(rPrevious \geq k) \mapsto (outServer \geq k)$

Proof: Simple cancellation of assertions below

Assertion $p \mapsto r \vee q$:

$$
\begin{aligned}
& rPrevious \geq k \\
= \quad & \{ \text{ predicate calculus } \} \\
& (rPrevious \geq k) \wedge ((outServer \geq k) \vee (outServer < k)) \\
\Rightarrow \quad & \{ \text{ predicate calculus } \} \\
& (outServer \geq k) \vee ((rPrevious \geq k) \wedge (outServer < k))
\end{aligned}
$$

Assertion $q \mapsto r$:

$$
\begin{aligned}
& (rPrevious \geq k) \wedge (outServer < k) \\
= \quad & \{ \text{ server invariant } \} \\
& ((S2 \vee S3) \wedge (outServer = k - 1)) \vee (S1 \wedge (outServer = k - 1))
\end{aligned}
$$

$\mapsto$ { simple cancellation with lemma 13 }

$(S2 \vee S3) \wedge (outServer = k - 1)$

$\mapsto$ { lemma 16 }

$outServer \geq k$

$\square$

LEMMA 20: NO TOKENS STUCK IN REPLY CHANNEL

$(sReply \geq k) \mapsto (rReply \geq k)$

Proof: Simple cancellation of assertions below

Assertion $p \mapsto r \vee q$:

$sReply \geq k$

$\Rightarrow$ { predicate calculus and lemma 17 }

$(rReply \geq k) \vee ((sRequest = sReply = k) \wedge (rReply = k - 1))$

Assertion $q \mapsto r$:

$(sRequest = sReply = k) \wedge (rReply = k - 1)$

$\Rightarrow$ { client invariant }

$C2 \wedge (sReply \geq k) \wedge (rReply = k - 1)$

$\mapsto$ { lemma 5 }

$C3 \wedge (rReply = k)$

$\Rightarrow$ { predicate calculus and math }

$rReply \geq k$

$\square$

LEMMA 21: CLIENT SENDS ITS TOKENS EVENTUALLY

$(rReply \geq k) \mapsto (sClientNext \geq k)$

Proof: Simple cancellation of assertions below

Assertion $p \mapsto r \vee q$:

$rReply \geq k$

$\Rightarrow$ { predicate calculus like in proof of previous lemma }

$(sClientNext \geq k) \vee ((rReply \geq k) \wedge (sClientNext{<}k))$

$\Rightarrow$ { client invariant }

$(sClientNext \geq k) \vee (C3 \wedge (sClientNext = k - 1))$

Assertion $q \mapsto r$:

$C3 \wedge (sClientNext = k - 1)$

$\mapsto$ { lemma 6 }

$C0 \wedge (sClientNext = k)$

$\Rightarrow$ { math and predicate calculus }

$sClientNext \geq k$

Simple cancellation completes the proof.

□

THEOREM 5: NO TOKENS DESTROYED OR STUCK

$(rPrevious \geq k) \mapsto (sNext \geq k)$

Proof:

$rPrevious \geq k$

$\mapsto$ { lemma 19 }

$outServer \geq k$

$\Rightarrow$ { definition $outServer \equiv sServerNext + sReply$ }

$< \exists i, j : i + j \geq k : (sReply = i) \wedge (sServerNext = j) >$

$\Rightarrow$ { pick i and j }

$(sReply \geq i) \wedge (sServerNext \geq j)$

$\mapsto$ { S-PSP on lemmas 20 and 9 }

$(rReply \geq i) \wedge (sServerNext \geq j)$

$\mapsto$ { S-PSP on lemmas 21 and 9 }

$(sClientNext \geq i) \wedge (sServerNext \geq j)$

$\Rightarrow$ { math and predicate calculus }

$(sClientNext + sServerNext) \geq (i + j)$

$\Rightarrow$ { definition $sNext \equiv sClientNext + sServerNext$ }

$sNext \geq (i + j)$

$\Rightarrow$ { $i + j \geq k$ and transitivity of $\geq$ }

$sNext \geq k$

$\square$

THEOREM 6: TOKENS CIRCULATE FOREVER

$true \mapsto rPrevious_i \geq k$

Proof:

The proof is by induction on k and i:

- base case $k = 1$, we induct on i:
  - base case $i = 0$, prove $true \mapsto rPrevious_0 \geq 0$
  - induction step, assuming $true \mapsto rPrevious_i \geq 0$
    prove $true \mapsto rPrevious_{i+1} \geq 0$ for $0 \leq i > N - 1$

  Both cases follows trivially from $rPrevious_i \geq 0$, (2) page 8.

- induction step, assuming $true \mapsto rPrevious_i \geq k$ for any i,
  prove $true \mapsto rPrevious_i \geq k + 1$:
  - base case $i = 0$, prove $true \mapsto rPrevious_0 \geq k + 1$
  - induction step, assuming $true \mapsto rPrevious_i \geq k + 1$
    prove $true \mapsto rPrevious_{i+1} \geq k + 1$ for $0 \leq i > N - 1$

• Induction step for k, assuming $true \mapsto rPrevious_i \geq k$ prove base case $i = 0$, $true \mapsto rPrevious_0 \geq k + 1$

Assertion $p \mapsto r \vee q$:

$true$

$\mapsto$ { induction hypothesis choosing $i = N - 1$ }

$rPrevious_{N-1} \geq k$

$\mapsto$ { theorem 5 }

$sNext_{N-1} \geq k$

$\Rightarrow$ { theorem 3 and predicate calculus }

$(rPrevious_0 \geq k+1) \vee (sNext_{N-1} = rPrevious_0 = sNext_0 = k)$

Assertion $q \mapsto r$:

$sNext_{N-1} = rPrevious_0 = sNext_0 = k$

$\Rightarrow$ { lemma 18 }

$S0_0 \wedge (sNext_{N-1} \geq k) \wedge (rPrevious_0 = k)$

$\Rightarrow$ { channel connection $sNext_{N-1} = sPrevious_0$, and $iPrevious_0 = 1$ }

$S0_0 \wedge (sPrevious_0 + iPrevious_0 \geq k+1) \wedge (rPrevious_0 = k)$

$\mapsto$ { lemma 11 with $j = k+1$ }

$S1_0 \wedge (rPrevious_0 = k+1)$

$\Rightarrow$ { math and predicate calculus }

$rPrevious_0 \geq k+1$

Simple cancellation completes the proof of the case.

• Induction step for k, induction step for i, assuming $true \mapsto rPrevious_i \geq k+1$ prove $true \mapsto rPrevious_{i+1} \geq k+1$

Assertion $p \mapsto r \vee q$:

$true$

$\mapsto$ { induction hypothesis }

$rPrevious_i \geq k+1$

$\mapsto$ { theorem 5 }

$sNext_i \geq k+1$

$\Rightarrow$ { predicate calculus and theorem 3 }

$(rPrevious_{i+1} \geq k+1) \vee$
$((sNext_i = k+1) \wedge (rPrevious_{i+1} = sNext_{i+1} = k))$

Assertion $q \mapsto r$:

$(sNext_i = k+1) \wedge (rPrevious_{i+1} = sNext_{i+1} = k)$

$\Rightarrow$ { lemma 18 }

$(sNext_i \geq k+1) \wedge S0_{i+1} \wedge (rPrevious_{i+1} = k)$

$\Rightarrow$ { channel connection $sNext_i = sPrevious_{i+1}$ and $iPrevious_{i+1} = 0$ }

$S0_{i+1} \wedge (sPrevious_{i+1} + iPrevious_{i+1} \geq k+1) \wedge (rPrevious_{i+1} = k)$

$\mapsto$ { lemma 11 with $j = k+1$ }

$S1_{i+1} \wedge (rPrevious_{i+1} = k+1)$

$\Rightarrow$ { math and predicate calculus }

$rPrevious_{i+1} \geq k+1$

Simple cancellation completes the proof of the case.

$\square$

LEMMA 22: NO SERVER WAITS FOREVER

$S0 \mapsto S1$

Proof:

$S0$

$\Rightarrow$ { predicate calculus }

$< \exists j :: S0 \wedge (rPrevious = j-1) >$

$\Rightarrow$ { pick j }

$S0 \wedge (rPrevious = j-1)$

$\Rightarrow$ { predicate calculus }

$true \wedge S0 \wedge (rPrevious = j-1)$

$\mapsto$ { PSP with theorem 6 and lemma 10 }

$((rPrevious \geq j) \wedge S0 \wedge (rPrevious = j-1)) \vee (S1 \wedge (rPrevious = j))$

$\Rightarrow$ { simplification }

$false \vee (S1 \wedge (rPrevious = j))$

$\Rightarrow$ { simplification }

$S1$

$\square$

LEMMA 23

$(S0 \vee S1 \vee S3) \mapsto S1$

Proof:

$$
\begin{array}{ll}
 & S0 \mapsto S1 \quad \text{(lemma 22)} \\
\Rightarrow & \{ \text{ simple disjunction with tautology } S1 \mapsto S1 \text{ } \} \\
 & (S0 \vee S1) \mapsto S1 \\
\Rightarrow & \{ \text{ simple disjunction with } S3 \mapsto S1 \text{ from subproof } \} \\
 & (S0 \vee S1 \vee S3) \mapsto S1
\end{array}
$$

Subproof:

$$
\begin{array}{ll}
 & S3 \\
\Rightarrow & \{ \text{ for some j } \} \\
 & S3 \wedge (sServerNext = j - 1) \\
\mapsto & \{ \text{ lemma 15 } \} \\
 & S0 \wedge (sServerNext = j) \\
\Rightarrow & \{ \text{ simplification } \} \\
 & S0 \\
\mapsto & \{ \text{ lemma 22 } \} \\
 & S1
\end{array}
$$

□

LEMMA 24: NO REQUEST IS STUCK IN REQUEST CHANNEL

$(sRequest \geq k) \mapsto (rRequest \geq k)$

Proof:

Assertion $p \mapsto r \vee b$:

$$
\begin{array}{ll}
 & sRequest \geq k \\
\Rightarrow & \{ \text{ lemma 17 } \} \\
 & (rRequest \geq k) \vee ((sRequest = k) \wedge (rRequest = sReply = k - 1))
\end{array}
$$

Assertion $b \mapsto r \vee q$:

$(sRequest = k) \wedge (rRequest = sReply = k-1)$

$\Rightarrow$ { server invariant }

$(S0 \vee S1 \vee S3) \wedge (sRequest \geq k) \wedge (rRequest = k-1)$

$\mapsto$ { PSP on lemma 23 and (7) page 8 }

$(rRequest \geq k) \vee (S1 \wedge (sRequest \geq k) \wedge (rRequest = k-1))$

Assertion $q \mapsto r$:

$S1 \wedge (sRequest \geq k) \wedge (rRequest = k-1)$

$\mapsto$ { lemma 12 }

$S2 \wedge (rRequest = k)$

$\Rightarrow$ { math and predicate calculus }

$rRequest \geq k$

Applying simple cancellation to the last two assertions with b for p gives $b \mapsto r$. Then applying simple cancellation to the first assertion and $b \mapsto r$ with b for q completes the proof.

□

LEMMA 25: EVERY REQUEST IS REPLIED EVENTUALLY

$(rRequest \geq k) \mapsto (sReply \geq k)$

Proof: Simple cancellation of assertions below

Assertion $p \mapsto r \vee q$:

$rRequest \geq k$

$\Rightarrow$ { lemma 17 }

$(sReply \geq k) \vee ((rRequest = k) \wedge (sReply = k-1))$

Assertion $q \mapsto r$:

$(rRequest = k) \wedge (sReply = k-1)$

$\Rightarrow$ { server invariant }

$S2 \wedge (sReply = k-1)$

$\mapsto$ { lemma 14 }

$S0 \wedge (sReply = k)$

$\Rightarrow$ { math and predicate calculus }

$sReply \geq k$

$\Box$

THEOREM 7: HUNGRY CLIENTS EAT EVENTUALLY

C1 $\mapsto$ C3

Proof:

$C1$

$\Rightarrow$ { client invariant }

$< \exists k :: C1 \wedge (sRequest = k) >$

$\Rightarrow$ { pick $k = j - 1$ }

$C1 \wedge (sRequest = j - 1)$

$\mapsto$ { lemma 3 }

$C2 \wedge (sRequest = j)$

$\Rightarrow$ { client invariant and weakening }

$(sRequest \geq j) \wedge C2 \wedge (rReply = j - 1)$

$\mapsto$ { PSP with result from subproof and lemma 4 }

$(C2 \wedge (sReply \geq j) \wedge (rReply = j - 1)) \vee (C3 \wedge (rReply = j))$

$\mapsto$ { simple cancellation with lemma 5 }

$C3 \wedge (rReply = j)$

$\Rightarrow$ { predicate calculus }

$C3$

Subproof:

$(sRequest \geq j) \mapsto (rRequest \geq j)$ (lemma 24)
$(rRequest \geq j) \mapsto (sReply \geq j)$ (lemma 25)

$\Rightarrow$ { transitivity of leadsto }

$(sRequest \geq j) \mapsto (sReply \geq j)$

$\Box$

11 Concluding Remarks

As mentioned in the introduction, the initial idea was to derive correct CC++ programs via stepwise refinement of UNITY programs. This idea was abandoned, since the final UNITY program seemed much more complex than the CC++ program itself. However, reasoning about control-points in the CC++ program is avoided, if the program is proved in a UNITY version. This might outweigh the disadvantage of the longer UNITY program.

We have made an effort to give a rigorous proof, however, the proof depends on the correctness of the basic lemmas and properties given for the server and the client. We are not completely satisfied with the derivation of these properties from the CC++ code and the characterization of the channels. A UNITY formalization of the channels would increase the formal level of these derivations.

CC++ channels allow the use of multiple senders and receivers and combined probe and receive actions. This is, however, not necessary to solve the mutual exclusion problem; passing the token directly from the client to the next server saves a single communication, but in most cases this has little influence on the overall efficience, and since there is only one receiver for the request, the probe and receive need not be executed atomically.

Martin gives three solutions to the mutual exclusion problem in [9]. The most efficient of these, "the reflecting privilege", uses a non-deterministic selection statement with a probe guard for each of the input channels. This type of statement is not available in CC++, but we can model it using a channel with multiple senders and a single receiver. This type of "merger" channel can also be used to implement the general dining philosophers problem.

Our future work will aim at a formalization of multiple senders/receivers channels and look at "the reflecting privilege" solution and the dining philosophers as examples of their use.

Acknowledgement

A warm thanks to my colleagues at Caltech for many useful discussions. In particular to my fellow students, Peter Hofstee, Marcel van der Goot, and Rustan Leino, and to Ralph Back and Jan van de Snepscheut who all found time to read and discuss an earlier version of this report, supplied many useful comments, and pointed out a number of interesting references. Also thanks to Marcel van der Goot for producing an environment for presenting the CC++ programs overnight. Last (but definitely not least) to Mani Chandy who advised me in my work and without whom this would never have come through.

Appendix: Proof Style and Proof Rules

The proof style is highly inspired by [6]. However, since we use it for reasoning about leadsto properties as well as ordinary predicates, some additional remarks are needed.

Due to the implication rule for the leadsto operator, and the transitivity of leadsto, any $a \Rightarrow b$ proof in the Dijkstra-Scholten style is also a $a \mapsto b$ proof. (And, obviously, since an $a = b$ proof, proofs both $a \Rightarrow b$ and $b \Rightarrow a$, it also proofs both $a \mapsto b$ and a $b \mapsto a$).

We have used the following rules in the proofs in the earlier sections:

- Transitivity and Simple Disjunction (from the disjunction rule)

$$\frac{p \mapsto q, q \mapsto r}{p \mapsto r} \quad \text{(Ltran)} \qquad \frac{p \mapsto q, p' \mapsto q}{p \vee p' \mapsto q} \quad \text{(Lsimd)}$$

- Implication and Progress-Safety-Progress

$$\frac{p \Rightarrow q}{p \mapsto q} \quad \text{(Limpl)} \qquad \frac{p \mapsto q, r \ \textbf{unless}\ b}{p \wedge r \mapsto (q \wedge r) \vee b} \quad \text{(Lpsp)}$$

- Simple Progress-Safety-Progress and Simple Cancellation

$$\frac{p \mapsto q, \textbf{stable}\quad r}{p \wedge r \mapsto q \wedge r} \quad \text{(Lspsp)} \qquad \frac{p \mapsto r \vee q, q \mapsto r}{p \mapsto r} \quad \text{(Lstep)}$$

Except for the Simple Cancellation rule which is proved below, all rules are proved in [4].

THEOREM 8: SIMPLE CANCELLATION

$$\frac{p \mapsto r \vee q, q \mapsto r}{p \mapsto r} \quad \text{(Lstep)}$$

Proof:

$$
\begin{array}{ll}
 & true \\
= & \{ \text{ predicate calculus } \} \\
 & r \Rightarrow r \\
\Rightarrow & \{ \text{ implication rule for leadsto } \} \\
 & r \mapsto r \\
\Rightarrow & \{ \text{ simple disjunction with assumption: } q \mapsto r \ \} \\
 & r \vee q \mapsto r \\
\Rightarrow & \{ \text{ transitivity with assumption: } p \mapsto r \vee q \ \} \\
 & p \mapsto r
\end{array}
$$

□

References

[1] 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.

[2] K. Mani Chandy and Carl Kesselman. The CC++ language definition. Technical Report Caltech-CS-TR-92-02, California Institute of Technology, 1992.

[3] K. Mani Chandy and Carl Kesselman. Communication and synchronization libraries in CC++. Technical Report Caltech-CS-TR-92-12, California Institute of Technology, 1992.

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

[5] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[6] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Text and Monographs in Computer Science. Springer-Verlag, 1990.

[7] G. LeLann. Distributed systems: Towards a formal approach. In *Proc. Information Processing 77*, pages 155–160, Amsterdam, 1977. North-Holland.

[8] Alain J. Martin. An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, 1981.

[9] Alain J. Martin. Distributed mutual exclusion on a ring of processes. *Science of Computer Programming*, 5:265–276, 1985.

[10] Jayadev Misra. Soundness of the substitution axiom. Notes on UNITY: 14-90, March 1990.

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

[12] A.J.M. van Gasteren and G. Tel. Comments on "on the proof of a distributed algorithm": Always-true is not invariant. *Information Processing Letters*, 35:277–279, September 1990.