Verification of a Distributed Computing System by Layered Proofs *

Cui Zhang† Brian R. Becker, Dave Peticolas,
Mark Heckman, Karl Levitt, and Ronald A. Olsson
Department of Computer Science, University of California, Davis, CA 95616

Abstract

This paper presents a technique for the verification of "full" distributed computing systems, building on the CLI stack which addresses verification of a layered sequential system. This paper also presents the application of our technique to the verification of a distributed system of three layers: a small high-level distributed programming language (microSR); a multiple processor architecture consisting of an instruction set and system calls; and a network interface. MicroSR programs are implemented by a compiler from microSR to the multiprocessor layer. System calls (for interprocess message passing) are implemented by network services. This work demonstrates that the correctness of a distributed program, most notably its interprocess communication, is verifiable through layers that guarantee the correctness of the compiled code that makes reference to operating system calls, of the operating system calls in terms of network calls, and of the network calls in terms of network transmission steps. The Cambridge HOL system is used for the specification and the proofs.

1 Introduction

System verification is very important for safety critical software development. This paper presents a technique for the verification of "full" distributed computing systems. The technique has been applied to the verification of a three layer system: (1) a high-level distributed programming language (microSR, a derivative of the SR language[2]); (2) a multiple processor architecture (MP machine) consisting of an instruction set and OS system calls; and (3) a network interface. MicroSR programs are implemented by a compiler that translates from microSR to MP machine code. System calls (for interprocess message passing) of the MP machine are implemented by network services. Through layered verification, it is guaranteed that a microSR program is correctly implemented by the assembly code that runs on the MP machine and that makes calls to network in support of interprocess message passing.

To formally specify a distributed language implementation, one must have a formal specification of the semantics of both the source and target languages. The challenge in the verification of distributed programming language implementations is dealing with this pair of specifications, each of which has its own model of concurrency, nondeterminism, and granularity of atomicity. Furthermore, to formally verify that high-level statements, especially message passing statements, can be ultimately implemented correctly by the network transmission steps, one must deal with multiple semantic specifications and multiple implementation proofs. To reduce the proof complexity of verifying the distributed language implementation, we believe that it is important to structure the semantic specification of each layer and the proof obligations that define the correctness conditions for the implementation.

Research on the verification of sequential language implementations has sharpened this belief. CLI has developed a *layered approach* and has shown its feasibility for a "full" sequential computing system verification: a compiler for a sequential language, an assembler and loader, and a microprocessor[3]. Windley has developed a method to specify instruction set architectures and a *generic interpreter model* for the layered proof of microprocessors[9]. Joyce has developed a HOL-based method to verify a *compiler* for a sequential programming language[7]. Curzon has developed a method to combine a derived programming logic with a verified *compiler* for an assembly language[5]. Research on the verification of distributed systems and programs has

*This work was sponsored by DARPA under contract USN N00014-93-1-1322 with the Office of Naval Research.

†The corresponding author may be reached at Department of Computer Science, California State University Sacramento, Sacramento, CA 95819-6021, zhangc@ecs.csus.edu

1060-3425/97 $10.00 © 1997 IEEE

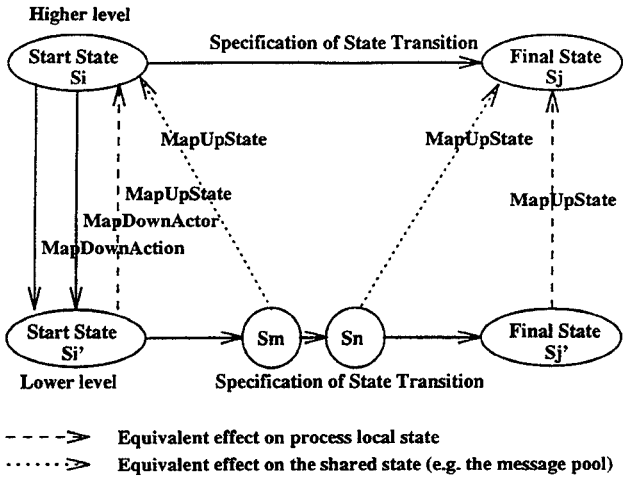

Figure 1: The correct implementation relationship between two adjacent layers.

shown that the traditional state transition model is extendible to address atomicity, concurrency, and nondeterminism[1, 4, 8].

As described in Section 2, our technique is based on the state transition model. We show that distributed "languages", whether actual higher-level programming languages, lower-level instruction sets for multi-computer systems, or a network interface wherein processes communicate through the exchange of packets, can be formalized under our state transition model. We use this model to account for the issues of atomicity, concurrency, and nondeterminism at all layers in a distributed computing system. We provide a framework, in terms of relations, for specifying the operational semantics of distributed programming languages (a very general view of languages as interfaces), so that semantic specifications at all layers have a similar structure. Moreover, a correctness definition is given which, for each pair of adjacent language semantics and mappings between them (i.e., the specification of language implementation), produces proof obligations corresponding to the correctness of the implementation. Figure 1 shows a schematic of the correct implementation relationship between two adjacent distributed language layers. Our technique not only makes the specification of distributed languages a step-by-step task, but also eases the verification effort for the distributed language implementation.

In addition to constructs basic to sequential languages, our microSR includes: (1) the asynchronous *Send* statement; (2) the synchronous *Receive* statement; and (3) the *Co* (co-begin) statement for specifying the concurrent execution of processes that communicate via message passing. The MP machine is an abstraction of an interface of multiple processors. In addition to conventional sequential instructions, two instructions for communication (system calls) are provided by the MP machine: asynchronous *SEND* and synchronous *RCV*. The network consists of a collection of network units accomplishing network operations, wherein processes communicate through the exchange of packets. Our current system is small, but non-trivial, resulting in non-trivial layered proofs. We believe that our technique has general applicability to the formal verification of larger layered distributed computing systems.

Our work is performed in HOL[6]. Section 2 describes our technique. Section 3 presents our proof of the microSR implementation by a compiler. Section 4 presents our proof of the implementation of MP machine system calls by the network services. Section 5 concludes the paper.

2 The Technique

2.1 The State Transition Model

All *primitive* statements at any given layer are atomic. Once a process starts executing a primitive statement, whether an intra-process statement or a communication primitive, no other process can influence that statement's execution or observe intermediate points of its execution. Thus, if two primitive statements, say C1 and C2, are executed concurrently in processes P1 and P2, the net effect is either that of C1 followed by C2, or of C2 followed by C1. Although we model the concurrent execution of two statements by two processes as a linearly ordered sequence of state transitions, the actual order in which selectable (i.e., eligible to execute) statements are executed is nondeterministic. With this view of atomicity, the behavior of a distributed program is modeled as a sequence of state transitions, each of which is accomplished by an atomic step. By structural induction, the execution of a composite statement (e.g., *if-then-else* statement) is actually an interleaving of the execution of its atomic components and the execution of atomic primitives of other processes.

Since the actual order in which eligible statements of different processes are executed is nondeterministic, multiple interleavings of the state transitions are possible. Thus, the model reflects concurrency in the semantics and the nondeterministic execution order of instructions. For the simple program below, the execution of synchronous "**Receive** mq2(v)" in

process P2 cannot be selectable until at least one message has been sent to the message queue mq2; this is true in all interleavings. The execution of "**Send** mq1(msg31)" in process P3 can occur either earlier or later than the execution of statements in other processes. This is because, by the language semantics, messages from the same sender to the same message queue have to be well ordered, but the order of messages from different senders is indeterminate.

A sample program:
(Process P1) ...**Send** mq2(msg11); **Send** mq2(msg12)...
(Process P2) ...**Receive** mq2(v)...
(Process P3) ...**Send** mq1(msg31)...

Some possible interleavings:
... **Send** mq2(msg11) **Receive** mq2(v)
Send mq2(msg12) **Send** mq1(msg31) ...
... **Send** mq2(msg11) **Send** mq1(msg31)
Receive mq2(v) **Send** mq2(msg12) ...
... **Send** mq1(msg31) **Send** mq2(msg11)
Send mq2(msg12) **Receive** mq2(v) ...

Our work has given careful attention to what will happen between the adjacent language systems when the higher-level language is implemented by the lower-level language. Because the atomicities of the two different layers have different granularities, a single atomic state transition at the higher language layer corresponds to multiple state transitions at the lower language layer. Among them, only those interleavings which exhibit equivalent effects will be allowed by a correct implementation of the higher-level language in terms of the lower-level language.

For the simple program above, the execution of "**Receive** mq2(v)" cannot be selectable until at least one message has been sent to the message queue mq2. However, when a microSR program is compiled to MP code, the MP code for "**Receive** mq2(v)" may begin its execution before the code for a microSR *Send* finishes execution, thus permitting concurrency in the implementation that is not apparent in the specification. For illustrative purpose, Figure 2 shows the implementation of a sequence of microSR *Send*s and *Receive*s, each requiring three MP instructions. The first stands for the preparation, the second is a *SEND* labeled S (or a *RCV* labeled R) to access a message queue, and the third one is for the clean-up. The execution of MP instruction "21", which corresponds to the "preparation" for the instruction R, begins before the execution of instruction "13", which corresponds to the "clean-up" after the instruction S. We allow this overlapping because of the finer atomicity and interleavings at the MP layer. However, the instruction R can be selectable

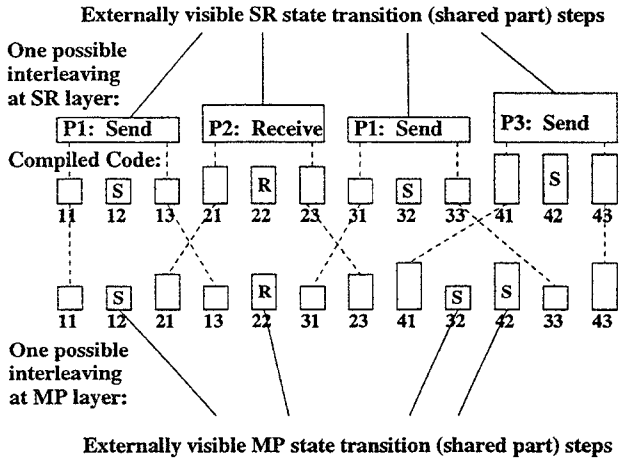

Figure 2: Interleavings at microSR and MP layers.

only when at least one message has been sent to the message queue by the instruction S.

Therefore, the following issues must be addressed in the semantic specification for a distributed language:

(1) the intra-process continuation which defines the decomposition of a composite statement into atomic transition steps;

(2) the intra-process sequencing of statements which defines the state transitions of two adjacent intra-process statements with potential interleaving of steps of other processes;

(3) the execution eligibility for the system-wide sequencing which defines the allowable interleavings, i.e., the synchronization of concurrent execution of multiple processes.

Furthermore, in order to verify the language implementation, the following issues must be addressed in the formalization of the language implementation:

(1) mappings of states, instructions, and processes, between a given pair of semantic specifications;

(2) the equivalence of interleavings at two layers;

(3) the proof obligation of the implementation correctness.

2.2 The Specification Framework for Distributed Languages

Since all possible interleavings are taken into account and since nondeterminism is permitted, the semantic specification for distributed languages is very different from its counterpart for sequential languages. Our specification aggregates definitions for *Syntax, State, Continuation, Selection, Mean-*

ing, and Co_Meaning. Our specification of the microSR language provides an example for describing how these definition should be made. The specifications of all three layers have a similar structure.

The abstract *Syntax* is recursively defined as a HOL type called *Stmt.* The semantic domain *State* is a configuration that combines (1) a collection of *Proc_local_states*, each of which is a process' local variable bindings; (2) a shared *Pool_state*, i.e., mappings from the name of a message queue to its message queue; and (3) a collection of *Threads*, each of which is a process' syntactic continuation. *Continuation* is the relation that formalizes the intra-process syntactic continuation, representing the "rest" of the computation that still has to be executed by the process in a given state. *Selection* is the relation that formalizes the execution eligibility of statements in a state, which actually defines the synchronization of concurrent execution of multiple processes formalized by the system-wide sequencing of valid interleavings. *Meaning* is a relation that specifies what change can happen to the state due to the execution of a selectable statement. The relation *Co_Meaning* specifies the meaning of a program in our language, i.e., the effect on the state of executing a set of concurrent processes.

2.2.1 The Continuation Relation

The relation *Continuation* has the signature *Thread* → *Thread* → *Stmt* → *Proc_local_state* → *Bool.* Primitive statements correspond to atomic state transition steps, while composite statements have to be decomposed to an intra-process sequence of primitive transitions, possibly interleaved with primitive transitions of other processes. The definition is operational, as it indicates how a primitive is "popped off" from the thread and how to decompose the composite statement recursively to generate a new thread that represents the syntactic continuation. The *Mbexp* is the meaning function for boolean expressions.

```
⊢ def Continuation oldthread newthread
                  (Send mq(iexp))(ls:Proc_local_state)=
                  (newthread = (TL oldthread))
Continuation oldthread newthread (stmt1; stmt2) ls =
(∃ls'. (Continuation (APPEND [stmt1;stmt2]
                             (TL oldthread))
                     (APPEND[stmt2](TL oldthread)
                             stmt1 ls) ∧
       Continuation (APPEND[stmt2](TL oldthread))
                    newthread stmt2 ls'))
...
```

2.2.2 The Selection Relation

In any given state, some statements are always selectable (such as *Assign*, and *Send*). Some are only conditionally selectable. For example, "**Receive** mq(v)" is selectable only when there is at least one unreceived message in the given message queue *mq.* As shown below, the relation *Selection*, which is of signature *State* → *Stmt* → *Proc_id* → *Bool*, is also recursively defined. Through *Selection*, we define the synchronization of concurrent execution, the system-wide sequencing of valid interleavings.

```
⊢ def
Selection (s:State) (Send mq(iexp)) (p:Proc_id) = True
Selection s (Receive mq(var)) p =
        (unreceived_msg mq (get_pool_state s) ≥ 1)
Selection s (stmt1 ; stmt2) p = Selection s stmt1 p
...
```

2.2.3 The Meaning Relation for Statements

The relation *Meaning*, which has the signature of *Stmt→State→State→Proc_id→Bool* , specifies the complete effect on any state by executing a selectable statement in a process. Thus, *Meaning statement state1 state2 p* is true if *state2* can be reached from *state1* by executing *statement* in process *p.*

An atomic state transition by a process is allowed only if its current statement satisfies relations on valid interleavings and continuation, i.e., *Selection* and *Continuation.* In the definition below, *m_Assign* indicates an effect on a process' local state by an assignment statement. *m_Send* only has an effect on the shared pool state, namely adding a single message to the appropriate message queue. *Miexp* is the meaning function for integer expressions. *mq_add_msg* characterizes the effect of inserting a new message into a given message queue, asserting that messages from the same sender are well ordered but the order of the messages from different senders is indeterminate.

```
⊢ def
m_atomic_stmt (v := iexp)(s1:State)
              (s2:State)(p:Proc_id) =
   Continuation (get_thread s1 p) (get_thread s2 p)
                (v := iexp)(get_local_state s1 p) ∧
   Selection s1 (v := iexp) p ∧ m_Assign v iexp s1 s2 p
m_atomic_stmt (Send mq(iexp)) s1 s2 p =
   Continuation (get_thread s1 p)(get_thread s2 p)
                (Send mq(iexp))(get_local_state s1 p)∧
   Selection s1 (Send mq(iexp)) p ∧
   m_Send mq iexp s1 s2 p
...
⊢ def
```

```
m_Assign (v:Var)(e:IExp)(s1:State)(s2:State)(p:Proc_id)=
   ((get_pool_state s1) = (get_pool_state s2)) ∧
   ((get_local_state s2 p) =
    (change_proc_state v (Miexp e (get_local_state s1 p))
                         (get_local_state s1 p)))
⊢ def
change_proc_state(v:Var)(data:Value)
                 (ls:Proc_local_state)(x:Var) =
       IF (x = v) THEN data ELSE (ls x)
⊢ def
m_Send (mq:msg_queue) (e:IExp)
        (s1:State) (s2:State) (p:Proc_id) =
   LET pool1 = (get_pool_state s1) IN
   ((get_local_state s1 p) = (get_local_state s2 p)) ∧
   ((get_pool_state s2) =
    (change_pool_state mq
           (mq_add_msg mq (mk_msg e s1 p) pool1)
                       pool1))
⊢ def
change_pool_state (mq:msg_queue)
  (new_mqvalue:mqvalue)
  (pool:Pool_state) (mq':msg_queue) =
  IF (mq' = mq) THEN new_mqvalue ELSE (pool mq')
```

The effect on a state by an intra-process sequence of state transitions, possibly interleaved with system-wide valid state transitions associated with other processes, is defined by the relation *m_proc_Seq*. The existence of possible interleavings is specified by relations *m_sys_interleaving* and *n_steps*. *atomic_of program (HD SL)* asserts that each step in the interleaving corresponds to an atomic statement of the given program. The *m_atomic_stmt (HD SL) (HD sl) (HD(TL sl)) (HD pl)* guarantees that each step in the interleaving satisfies system-wide execution eligibility and the continuation relation in its own process. *m_proc_Seq* also asserts that other processes' effects on the state will not change this given process' local state and its effect on the shared message pool. *EL* is a list element selector.

```
⊢ def
m_proc_Seq(trans1,trans2: State→State→Proc_id→Bool)
          (s1,s2:State)(p:Proc_id)=
∃(s3:State)(s4:State)(program:(Stmt)list) .
m_sys_interleaving s3 s4 program ∧
((get_local_state s3 p)=(get_local_state s4 p)) ∧
((proc_effect_on_pool s3 p)=(proc_effect_on_pool s4 p))∧
          trans1 s1 s3 p ∧ trans2 s4 s2 p
⊢ def
m_sys_interleaving (s1:State) (s2:State)
                   (program:(Stmt)list) =
∃ (n:num)(SL:(Stmt)list)(sl:(State)list)(pl:(Proc_id)list).
((HD sl) = s1) ∧ ((EL (LENGTH sl) sl) = s2) ∧
((LENGTH SL) = n) ∧ ((LENGTH pl) = n) ∧
((LENGTH sl) = n+1)∧n_steps n SL sl pl program s1 s2
⊢ def
n_steps 0 SL sl pl program s1 s2 = (s1 = s2)
n_steps (SUC n) SL sl pl program s1 s2 =
  ((HD sl) = s1) ∧ ((EL (LENGTH sl) sl) = s2) ∧
  atomic_of program (HD SL) ∧
  m_atomic_stmt (HD SL) (HD sl)
                (HD(TL sl)) (HD pl) ∧
  n_steps n (TL SL) (TL sl) (TL pl)
          program (HD(TL sl)) s2
```

The definition of the meaning relation for composite statements depends on the meaning relations of their component statements. This means that the state transition accomplished by a composite statement is reduced recursively to state transitions accomplished by its component statements which must all satisfy relations on valid interleavings as well. Note that, because of this reduction, the arguments *trans1* and *trans2* in the definition of *m_proc_Seq* are of the signature *State* → *State* → *Proc_id* → *Bool*, rather than simply of the type *Stmt* for statements. The definition of the *Meaning* relation shows what these *trans1* and *trans2* really represent and how relations like *m_proc_Seq* are used. In this way, the concurrency and nondeterminism are completely specified.

```
⊢ def
Meaning (Send mq(e)) s1 s2 p =
        m_atomic_stmt (Send mq(e)) s1 s2 p
Meaning (stmt1 ; stmt2) s1 s2 p =
        Continuation (get_thread s1 p)(get_thread s2 p)
             (stmt1;stmt2)(get_local_state s1 p)∧
        Selection s1 (stmt1 ; stmt2) p ∧
        m_proc_Seq (Meaning stmt1)
                   (Meaning stmt2) s1 s2 p
...
```

2.2.4 The Co_Meaning Relation

A program in our language is represented by a *Co* statement which consists of a set of concurrently executed processes, each of which is defined by a process id and a statement with the recursive structure of statement sequence. The meaning of a *Co* statement is specified by the relation called *Co_Meaning*. This relation has the signature *(Stmt)list* → *(Proc_id)list* → *State* → *State* → *Bool*. This relation is satisfied if the final state *s2* can be reached by beginning the concurrent execution of the program in state *s1*, where the *ith* process executes the *ith* branch of the *Co* statement. Because the concurrent execution of atomic statements is modeled as a linearly ordered sequence of state transitions, and the execution of a composite statement is modeled as the sequential

state transitions of its own atomic components interleaved with other process' atomic transitions, relation *Co_Meaning* specifies completely "what can happen" in the state by the concurrent execution of a distributed program.

```
⊢ def
Co_Meaning (program:(Stmt)list) (pl:(Proc_id)list)
           (s1:State) (s2:State) =
∀ (i:num) . ∃ (s1':state) (s2':state) .
Meaning (EL i program) s1' s2' (EL i pl) ∧
m_sys_interleaving s1 s1' program ∧
m_sys_interleaving s2' s2 program ∧
(get_local_state s1 (EL i pl) =
 get_local_state s1' (EL i pl)) ∧
(get_local_state s2' (EL i pl) =
 get_local_state s2 (EL i pl)) ∧
(proc_effect_on_pool s1 (EL i pl) =
 proc_effect_on_pool s1' (EL i pl)) ∧
(proc_effect_on_pool s2' (EL i pl) =
 proc_effect_on_pool s2 (EL i pl))
```

2.3 The Proof Obligation for Language Implementation

To prove the correctness of an implementation of a higher-level distributed language in terms of a lower-level distributed language, we have formalized the concept of the correct implementation relationship of two adjacent language layers. As shown in Figure 1, it is necessary to specify three mappings between two adjacent language layers to represent fully the language implementation. Its correctness has to be verified with respect to the pair of operational semantics. The "execution" of the "generated" lower-level instructions has to be proved to correctly implement the meaning of the corresponding higher-level instruction with respect to the corresponding start and final states at the higher layer. The proof obligation *Stmt_implemented_correct* for the implementation correctness and the relation *Equivalent_interleaving* for the equivalence of interleavings at two adjacent layers are specified below. Notice that, in the implementation correctness proof, the premise of the obligation has to be proved as a theorem first.

Because of the nondeterminism at two layers and the finer atomicity and interleavings at the lower layer, the correspondence of start states and final states at two layers are not unique. The "coincidence" of these states has been taken into account in the definitions below. As shown in Figure 1, the equivalence of interleavings at two layers is specified with respect to their equivalent effects on states. The equivalence of effects on the local states of processes is defined with respect to start and final states at both layers, while the equivalence of effects on the shared states (e.g., message pool states) is defined with respect to start and final states at the higher layer and two intermediate states at the lower layer. The two intermediate states indicate the critical state transition step at the lower layer where a change to the shared pool takes place. Since the lower layer allows finer atomicity and interleavings, this pair of intermediate states is also actually not unique. In the correctness proof, the existence of such a pair of intermediate states has to be shown for the implementation of each given statement.

```
MapDownAction: high_Stmt → low_Stmt
MapUpState: Low_State → high_State
MapDownActor: high_Proc_id → low_Proc_id

⊢ def
Stmt_implemented_correct (stmt:high_Stmt)
      (si, sj: high_State) (si', sj': low_State)
      (p:high_Proc_id) =
(low_m_proc_Seq (MapDownAction stmt)
                si' sj' (MapDownActor p)
 ⇒ ∃ (sm, sn: low_State).
   Equivalent_interleaving stmt
                           si sj si' sj' sm sn p)
⇒
high_Meaning stmt si sj p

⊢ def
Equivalent_interleaving stmt si sj si' sj' sm sn p =
ordered si' sm sn sj' ∧
(high_get_local_state si p =
 high_get_local_state(MapUpState si')p) ∧
(high_get_local_state sj p =
 high_get_local_state(MapUpState sj')p) ∧
(high_get_pool si = high_get_pool (MapUpState sm)) ∧
(high_get_pool sj = high_get_pool (MapUpState sn))
⇒
high_effect_on_local stmt (high_get_local_state si p)
                     (high_get_local_state sj p) p ∧
high_effect_on_pool stmt (high_get_pool si)
                     (high_get_pool sj) p
```

3 Verification of microSR Language Implementation

As mentioned in Section 1, a microSR program is compiled into MP machine code, where the MP machine is a specification of a basic multi-processor machine. Architecturally, the MP machine is viewed as a collection of simple RISC-based microprocessors (called VMachines) linked by a fully-connected point-to-point network.

3.1 MP Machine Specification

In accordance with our overall semantic specification framework, we formalize the semantics of instruction execution at the MP layer by generating the definitions of Section 2.2. Following the framework we have devised, the MP machine specification defines the *Syntax*, the *State*, and the accompanying relations. These are structurally very similar to those used by the microSR specification, differing only in the definition details reflecting the concurrency and nondeterminism at the MP layer.

The *Syntax* of the language at the MP layer is simply the instruction set defined by the VMachines—which consists of a small (14) set of basic instructions like ADD, JMP, LD, and STO—augmented to include two system calls which provide inter-processor communication. These two system calls are SEND, which asynchronously sends a message from a processor to a specific message queue, and RCV, which synchronously receives a message from a specific queue.

The *State* at this layer contains the same three major components as at the microSR layer, but with different formats to reflect the differing needs of the two layers. Each *local state* within the MP state is represented by the register set and memory contained within the corresponding VMachine. The *shared message pool* consists of a collection of message queues which are manipulated by the SEND and RCV system calls mentioned above. Each *thread* contains the code to execute on a particular VMachine, and a program counter for that VMachine.

In most cases, the *Continuation* relation can be satisfied with any two threads that contain the same code and consecutive values for the PC. The JMP and JZ instructions are the only exceptions to this rule because they modify the program counter. All of the VM instructions and the SEND system call satisfy the *Selection* relation in any given state because none need to block. The RCV system call, however, is only selectable if a message exists on the desired queue.

The *Meaning* relation for single instructions is made simple at the MP layer because all of the instructions are viewed as being atomic. However, because we have multiple processors executing code in a nondeterministic order, we must allow for instructions from multiple processors to interleave their execution. Thus, as explained in section 2.2, we must specify the effects of both intra-process and inter-process sequencing on the MP state. For this purpose, *mp_m_proc_Seq* is defined to formalize the execution of a sequence of atomic MP instructions by one processor interleaved with allowable steps of other processors.

3.2 Mappings Between the MP Layer and the microSR Layer

As mentioned in Section 2.3, one of the major components of an implementation proof is to establish mappings between the two layers under discussion. These mappings work to transform the code (*Action*), the *State*, and the process (*Actor*).

Mapping up the *State* essentially involves a translation of the data stored in the MP state to the equivalent structures used by the microSR state. Most notable about this mapping is that the VMachine memory, represented as a list, is transformed into the variable bindings used at the microSR layer, represented as a function. Furthermore, the pool (and, thus, all of the messages within it) must be manipulated into the more abstract form used by microSR. Mapping the *Actor* is an function on the process IDs.

Mapping down the code is more complicated. For this, we have formalized a compiler for microSR, implemented within HOL. This compiler transforms each statement in the microSR program into the appropriate sequence of instructions to be executed by the MP machine. Furthermore, microSR statements in different microSR processes are transformed to sequences of MP instructions which are to be concurrently executed by their corresponding processors. Therefore, as indicated previously, the execution of MP code can be interleaved with other code executed by other processors. For our initial compiler, little attempt at optimization has been attempted.

3.3 microSR Implementation Correctness Proof

Figure 3 illustrates the proof obligation of the implementation of microSR. The general form of the theorems for the correctness of the microSR implementation is given below, where the srstmt can be any of the statements defined by microSR:

```
⊢ SR_implemented_correct (srstmt: Stmt)
      (si, sj: SR_State) (si', sj': MP_State)
      (p: SR_Proc_id) =
(mp_m_proc_Seq(sr_mp_MapDownCode srstmt)
                si' sj' (sr_mp_MapDownPid p)
 ⇒ ∃ (sm:MP_State)(sn:MP_State).
   MP_Equivalent_Interleaving srstmt
                              si sj si' sj' sm sn p)
⇒
```

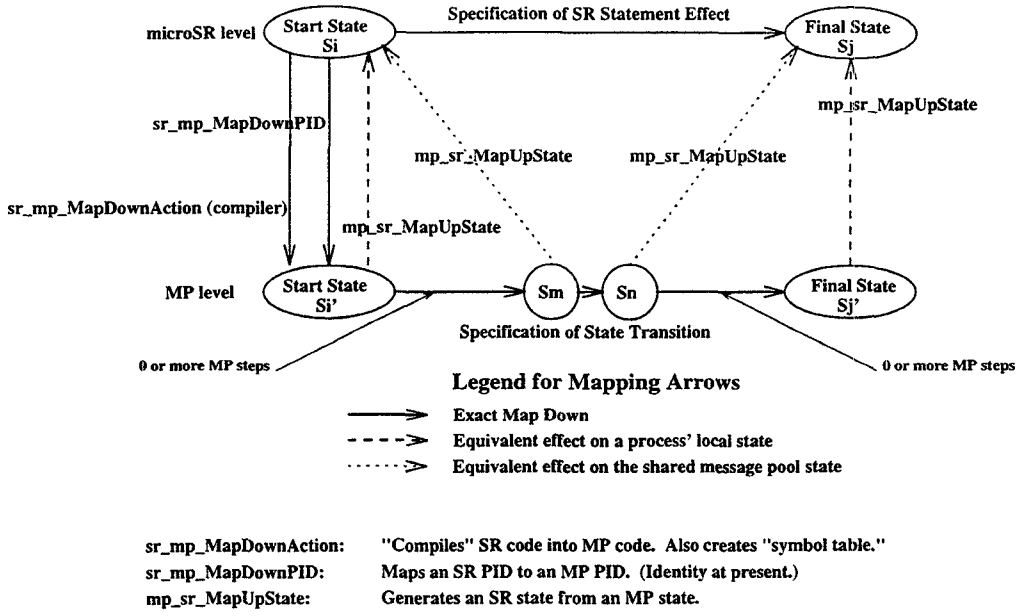

Figure 3: A Verified microSR Implementation

```
SR_Meaning srstmt si sj p

⊢ def
MP_Equivalent_Interleaving srstmt
                          si sj si' sj' sm sn p =
mp_ordered si' sm sn sj' ∧
(SR_local_state si p =
 get_sr_local (mp_sr_MapUpLocals si') p) ∧
(SR_local_state sj p =
 get_sr_local (mp_sr_MapUpLocals sj') p) ∧
(SR_get_pool_state si = mp_sr_MapUpPool sm) ∧
(SR_get_pool_state sj = mp_sr_MapUpPool sn)
⇒
SR_effect_on_locals stmt (SR_local_state si)
                    (SR_local_state sj) p ∧
SR_effect_on_pool stmt (SR_get_pool_state si)
                   (SR_get_pool_state sj) p
```

4 Verification of MP Layer System Calls

4.1 Network Specification

The SEND and RCV system calls of the MP layer are implemented by the network layer. The network layer is modeled as a collection of interconnected network interface units, or NIUs, with one NIU connected to each processor in the MP machine. The current state of the network communication channels is represented by an "in-transit" structure. The network in-transit structure is similar to an MP layer message pool, but with network packets instead of MP messages as the basic elemental units.

For each NIU, the in-transit structure contains a list of network packets that have been sent to that NIU as well as a list of packets which have been received by that NIU and transferred to the MP machine. The packets which have been sent to an NIU but not transferred to the MP machine include packets which are still being transmitted over the network as well as packets which have been received by an NIU, but not yet transferred to the MP machine. Each NIU has a local state which consists of a sequence of network operations which that NIU is scheduled to perform and a count of messages sent from that NIU. A complete network state consists of the shared in-transit state and the local state of each NIU.

Similar to the microSR and MP layers, the NIU operations are specified as a distributed programming language where the statements in the language correspond to network operations performed by a given NIU. The network programming language consists of seven atomic statements. The simplest statement is the "skip" or "no-op" statement which does not change the network state in any way. Its use is explained in Section 4.2.

MP SEND and RCV are both implemented by a sequence of three network statements (operations): an initialization operation, a transfer operation, and a termination operation which completes the system call. In the case of the SEND system call, the trans-

fer operation transmits a packet containing the MP message over the network. In the RCV system call, the transfer operation extracts a message from a received packet and passes it to the MP machine.

The meaning of each statement in the language is represented as a predicate over state transitions, as described in Section 2. The meaning of a sequence of network operations performed by a given NIU is an interleaving of that NIU's operations with operations performed by other network units. The meaning of an interleaving of network operations is a sequence of state transitions determined by the corresponding operation predicates. The operation of the entire network is determined by the predicate net_m_proc_Seq which selects the set of all valid interleavings of network operations by individual NIUs.

As explained in Section 2, the task of specifying the meaning of the system operation is aided by the use of the *Continuation* and *Selection* predicates. Since the network language has no mechanism for looping, the *Continuation* of a network operation is simply the "tail" of the list of operations that a given NIU is executing. The *Selection* predicate is identically true for all network statements except the transfer operation which is used in the MP Receive implementation. This network operation is only selectable when there are packets which have been received, but not transferred to the MP machine.

4.2 The Mappings between the Network Layer and the MP Layer

To formalize the implementation of MP system calls, there are three mappings between the Network Layer and the MP Layer: a mapping from MP machine process id's to network NIU id's; a mapping from MP machine instructions to network operations; and a mapping from a network state to an MP machine state.

The MP SEND and RCV system calls are each mapped into a sequence of three network operations by the mapping function NetMapDownInst. All other MP instructions are mapped to network "skip" instructions, since they have no effect on the MP communications state. It is important to note that the mapping between MP system calls and network operations is dynamic. Thus, the mapping function is an interpreter rather than a compiler. This removes the necessity for loops in the distributed network programming language. In practice, this is not a problem since the mapping will actually take place dynamically by executing the appropriate system call, rather than by static compilation.

Unlike the MP layer, which implements the entire semantics of the next higher layer (the microSR language), the network layer only needs to implement a subset of the MP layer (the communications portion of the MP specification). Thus, the entire network state is mapped up to an MP message pool, which is the shared portion of an MP machine state. The local portion of the MP machine state, which consists of the states of the individual microprocessors, is not determined by the state of the network.

4.3 Proof of Network Correctness

A diagram of the proof obligation for the MP layer implementation is shown in Figure 4. The network proof is comprised of two top-level theorems, one for the MP SEND implementation and one for MP RCV. Both theorems have the form shown below:

```
⊢ MP_implemented_correct (mpstmt:MP_Stmt)
      (si, sj: MP_State) (si', sj': net_State)
      (p:MP_Proc_id) =
(net_m_proc_Seq (NetMapDownInst mpstmt)
                si' sj' (NetMapDownPid p)
 ⇒ ∃ (sm:net_State)(sn:net_State).
   Net_Equivalent_interleavingmpstmt
                                si sj si' sj' sm sn p)
⇒
MP_Meaning mpstmt si sj p
```

```
⊢ def
Net_Equivalent_interleaving stmt si sj si' sj' sm sn p =
ordered si' sm sn sj' ∧
(MP_get_pool si = NetMapUpMsgPool sm) ∧
(MP_get_pool sj = NetMapUpMsgPool sn)
⇒
MP_effect_on_pool stmt (MP_get_pool si)
                       (MP_get_pool sj) p
```

5 Conclusions

Verifying the correctness of a multi-layered distributed computing system, from a high-level distributed programming language to a network interface, is a very difficult task. Our technique makes this task more tractable by providing a generic specification framework and a generic proof obligation. We have applied our technique to verify the implementation of microSR through layered proofs of a "full" distributed computing system that guarantee the correctness of the compiled code that makes reference to operating system calls, guarantee the correctness of the operating system calls in terms of

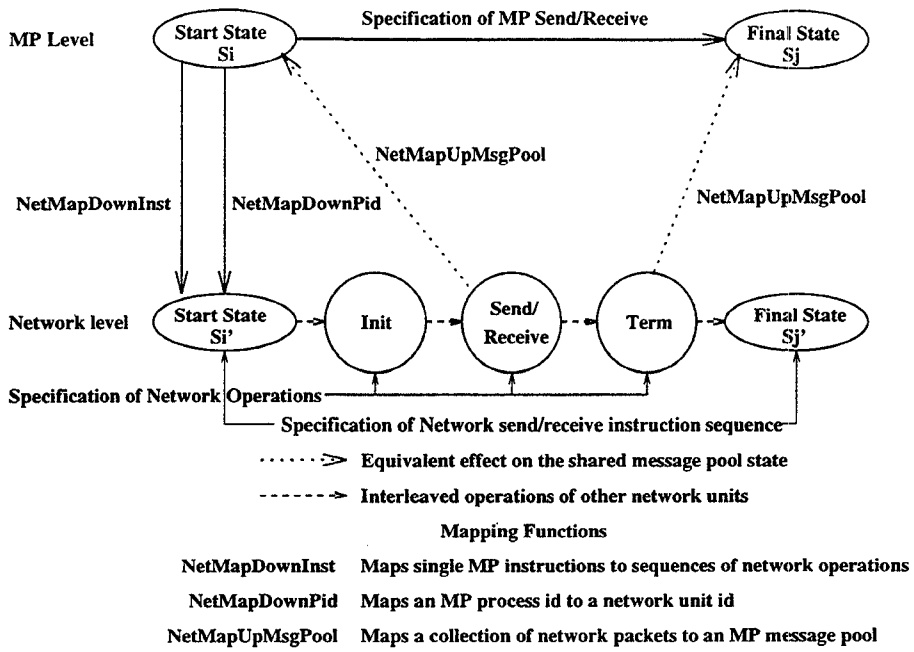

Figure 4: A verified MP system call implementation.

network calls, and guarantee the correctness of network calls in terms of network transmission steps.

Our current system is small, but non-trivial, resulting in non-trivial layered proofs. We are currently working on evolving our small layered distributed system to a larger system through additional functionality at each layer. One of these extensions will be a more realistic operating system providing dynamic process creation. We intend to demonstrate that the layered system proof can evolve in unison with the system design, and that our technique is applicable to the verification of larger distributed systems.

References

[1] M. Abadi and L. Lamport, 'The Existence of Refinement Mappings', *Theoretical Computer Science*, Vol. 82 (1992), pp. 253-284.

[2] G. R. Andrews and R. A. Olsson, *The SR Programming Language: Concurrency in Practice*, (Benjamin/Cummings Publishing Company, Inc. Redwood City, CA, 1993).

[3] W. R. Bevier, W. A. Hunt, J. S. Moore, and W. D. Young, 'An Approach to Systems Verification', *Journal of Automated Reasoning*, 5 (1989) 411–428.

[4] M. Chandy and J. Misra, *Parallel Program Design: A Foundation of Programming Logic*, (Addison-Wesley Publishing Company, Inc. 1988).

[5] P. Curzon, 'Deriving Correctness Properties of Compiled Code', in: *Higher Order Logic Theorem Proving and Its Applications*, (IFIP Transactions, A-20, North-Holland, 1993) pp327–346.

[6] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, (Cambridge University Press, Cambridge, 1993).

[7] J. J. Joyce, 'Totally Verified Systems: Linking Verified Software to Verified Hardware'. In: *Specification, Verification and synthesis: Mathematical Aspects*, edited by M. Leeser and G. Brown, (Springer-Verlag, 1989), pp. 177-201.

[8] A. U. Shankar, 'An Introduction to Assertional Reasoning for Concurrent Systems', *ACM Computing Surveys*, Vol.25, No.3 (September 1993), pp. 225-262.

[9] P. J. Windley, 'A Theory of Generic Interpreters', in: *Correct Hardware Design and Verification Methods*, No. 683 in LNCS (Springer-Verlag, 1993), pp. 122-134.