LVT: A Layered Verification Technique for Distributed Computing Systems

Cui Zhang[⋆], Brian R. Becker[†], Dave Peticolas[†],
Ronald A. Olsson[†], and Karl N. Levitt[†]

[⋆]*Department of Computer Science, California State University, Sacramento, CA 95819-6021*

Email: zhangc@ecs.csus.edu

[†]*Department of Computer Science, University of California, Davis, CA 95616-8562*

Email: {beckerb, peticola, levitt, olsson}@cs.ucdavis.edu

Abstract

This paper presents a *layered verification technique*, called LVT, for the verification of distributed computing systems with multiple component layers. Each lower layer in such a system provides services in support of functionality of the higher layer. By taking a very general view of programming languages as interfaces of systems, LVT treats each layer in a distributed computing system as a distributed programming language. Each relatively higher-level language in the computing system is implemented in terms of a lower-level language. The verification of each layer in a distributed computing system can then be viewed as the verification of implementation correctness for a distributed language. This paper also presents the application of LVT to the verification of a distributed computing system, which has three layers: a small high-level distributed programming language; a multiple processor architecture consisting of an instruction set and system calls for inter-process message passing; and a network interface. Programs in the high-level language are implemented by a compiler mapping from the language layer to the multiprocessor layer. System calls are implemented by network services. LVT and its application demonstrate that the correct execution of a distributed program, most notably its inter-process communication, is verifiable through layers. The verified layers 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 specification and verification involved are carried out by using the Cambridge HOL theorem proving system.

Keywords: distributed computing systems, system verification, implementation correctness, layered proofs.

1 Introduction

Many distributed computer applications, especially safety critical and financial critical applications, require a guarantee of the correctness of the entire distributed computing system. The applications are written in a high-level distributed programming language provided as the interface of the computing system. The correct execution of the applications requires the correctness of the individual components in the computing system. *System verification* has become increasingly important because the required guarantee of correctness cannot be established without the verification of multiple component layers in distributed computing systems. Typically such distributed computing systems include a language compiler, an operating system, a set of microprocessors, and a communication network that connects those microprocessors. Each lower layer provides services in support of functionality of the higher layer. Therefore, for any distributed application developed in a distributed programming language, the required correctness can only be fully assured by a formal verification of the entire distributed computing system that includes multiple verification activities: formal verification of a compiler for a distributed programming language, formal verification of an operating system, formal verification of a microprocessor, and formal verification of a communication network. However, as shown by the authors' previous experience (Zhang, Shaw, Heckman, Benson, Archer, Levitt, and Olsson, 1994), it is challenging to carry out these verification activities, and, without a systematic methodology, it is difficult to compose all the proofs at different layers to establish the correctness of an entire distributed computing system.

This paper presents a *layered verification technique*, called LVT, for the verification of distributed computing systems. Building on CLI's methodology for verifying a layered sequential system (Bevier, Hunt, Moore, and Young, 1989), LVT addresses the following issues.

- How to generically structure multiple layers of specifications to make the verifications of multiple layers tractable;
- How to generically formalize the proof obligation for the implementation correctness to ease the verifications at different layers and their compositions;

- How to use a theorem proving system to fulfill the above for attaining the level of rigor and reliance.

By taking a very general view of programming languages as interfaces of systems, LVT treats each layer in a distributed computing system as a distributed programming language. Each relatively higher-level programming language in the computing system is implemented in terms of a lower-level language, with the functionality of the higher-level language supported by that of the lower-level language. The verification for each layer in a distributed computing system can then be viewed as the verification of implementation correctness for a distributed language with respect to an adjacent pair of languages in the distributed computing system. Therefore, with LVT, the verification of a distributed computing system guarantees the implementation correctness of its interface high-level distributed programming language with respect to the complete layered system.

To formally verify a language implementation, one must have a formal specification of the semantics of both the source (higher-level) and target (lower-level) 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 complexity of verifying the distributed language implementation, LVT structures, with genericity, the semantic specification of each layer and the proof obligation between each pair of adjacent layers. The proof obligation defines what has to be proved for establishing the language implementation correctness.

LVT is based on the state transition model. This model is used to account for the issues of atomicity, concurrency, and nondeterminism at all layers in a distributed computing system. This paper also shows 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 the state transition model. With this very general view of languages as interfaces, LVT provides a generic framework, in terms of relations, for specifying the operational semantics of distributed programming languages. Following this framework, semantic specification at all layers in the distributed computing system can have a similar structure to address their common semantic aspects at all layers, while semantic specifications differ in details to reflect unique semantic characteristics at each layer. Moreover, for each pair of adjacent language semantics and an implementation relationship between them, LVT provides a generic proof obligation corresponding to the correctness of the implementation. Thus, LVT not only makes the semantic specification of distributed languages a step-by-step task, but it also eases the verification effort for distributed computing systems with multiple layers.

To show the feasibility of LVT, this paper also presents the application of LVT to the verification of a three layer system: a high-level distributed programming language called microSR, a derivative of the SR language (Andrews and Olsson, 1993); a multiple processor architecture called the MP machine, consisting of an instruction set and OS system calls for inter-process message passing; and a network interface. MicroSR programs are implemented by a compiler that translates from microSR programs to MP machine code. System calls 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 the network in support of inter-process message passing. Even though LVT is independent of the theorem proving system used, the specification and verification in this paper are carried out by using the Cambridge Higher Order Logic (HOL) theorem proving system (Gordon and Melham, 1993).

The layers in the distributed computing system discussed in this paper are simplified, yet representative of those found in a real distributed computing system. In addition to constructs basic to sequential languages, the microSR language includes: the asynchronous *Send* statement; the synchronous *Receive* statement; and 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, i.e., system calls, are provided by the MP machine: asynchronous *SEND* and synchronous *RCV*. *SEND* asynchronously sends a message to a specific message queue over an abstract network and *RCV* synchronously receives a message from a specific message queue. The network consists of a collection of network units accomplishing network operations, wherein processes communicate through the exchange of packets. The current distributed computing system is small, but non-trivial, resulting in non-trivial layered proofs. LVT itself has a general applicability to the formal verification of larger layered distributed computing systems.

Section 2 discusses the background for this work and the related research in this area. Section 3 describes LVT, including the state transition model, the semantic specification framework, and the proof obligation for the language implementation correctness. Section 4 presents the verification of the compiler translation of microSR. Section 5 presents the verification of the network service implementation of MP machine system calls. Section 6 concludes the paper.

2 Background and Related Work

2.1 System Verification

The work presented is inspired by the research on layered system verification. A layered system verification in general includes the following steps.

- The vertical decomposition of the system to be verified into layers of system components to reduce the complexity in the verification caused by the significant semantic gap between the high-level system interface language and the very low-level hardware machine;
- The verification of all the component layers;
- The composition of all the verified layers to establish the correctness of the entire system being verified.

CLI was the first to develop a methodology for machine-checked system verification of sequential computing systems (Bevier, Hunt, Moore, and Young, 1989; Good and Young, 1991). The methodology has been applied to the verification of a sequential computing system, called the *CLI short stack*, from a high-level programming language to hardware. The formal verification includes the verification of the correctness of a compiler from the high-level programming language Micro-Gypsy to the Piton assembly language, of a link-assembler from Piton to the FM8502 machine code, and of a gate-level register transfer model. The semantic consistence involved in the composition of proofs is addressed by using the same specification between any two adjacent layers when proving the correctness of those two layers. The work shows the feasibility of layered verification of sequential computing systems. In addition, since the semantics of Micro-Gypsy is guaranteed by the entire verified computing system, this work can also be viewed as an implementation verification for the high-level programming language through the layered verification of the computing system. By changing the component layer FM8502 to FM9001 in the short stack, CLI has also demonstrated that the specification and verification for the upper layers in the original stack can be reused to compose a system verification for the changed or new stack (Good, Kaufmann, and Moore, 1992).

LVT is built on the CLI methodology for verifying a layered sequential computing system. The technique presented in this paper significantly generalizes CLI's approach by addressing issues involved in the verification of distributed computing system with concurrency and nondeterminism in semantics at all the layers. The development of LVT is also inspired by the research of Windley (Windley, 1993), of Joyce (Joyce, 1989), and of Curzon (Curzon, 1993) on structuring specifications and proof obligations in sequential system verification. Windley has developed a layered method for the verification of instruction set architectures and a *generic interpreter model* that provide a generic approach to the semantic specification of all layers and to the specification of the proof obligation between each pair of adjacent layers. This work eases the effort of the layered verification of single and sequential micro-processors. Joyce has developed a method to verify a *compiler* for a sequential programming language. Curzon has developed a method to combine a derived programming logic with a verified *compiler* for an assembly language. These approaches use HOL in their language specifications (syntactic specification and semantic specification) and verifications. Specifically, they make good use of HOL features in recursive definitions and in theorem proving by structural induction on recursively defined structures. LVT extends these approaches for sequential systems to the verification of distributed computing systems, aiming at reducing the complexity in the layered verification and making the specifications and proofs involved more manageable.

2.2 Work based on State Transition

Research on the verification of distributed programs and on concurrent systems has shown that the traditional state transition model is extendible to address atomicity, concurrency, and nondeterminism (Chandy and Misra, 1988; Francez, 1992; Abadi and Lamport, 1992; Shankar, 1993). Among these works, the work by Chandy and Misra and the work by Francez address the verification of distributed programs written in high-level programming languages, a given layer at the top of a distributed computing system. However, they would not be directly applicable to the verification at lower layers in a distributed computing system, e.g., the verification at microprocessor layer. In addition, these works are not intended to address the issue of formal verification of implementation correctness. The work by Shankar and the other approaches to assertional reasoning of concurrent systems are covered by Shankar's tutorial (Shankar 1993). These approaches show how to prove that a concurrent system satisfies a set of desired properties by using safety assertions and progress assertions based on temporal logic. These approaches can be applied to specify and reason about concurrent systems at various levels. However, they were not developed to support and facilitate verification of implementation correctness involving multiple levels. It is not clear, therefore, exactly how to extend these approaches for such verification. Abadi and Lamport's methodology is intended for the verification of refinement of concurrent systems that are usually developed as an application system on top of a distributed computing system. The refinement of a concurrent system is verified with respect to the specifications of their desired behaviors and properties. No matter how many steps of refinement are involved, the same concurrent system with different representations is being verified. When applying this method, one can reasonably assume the availability of the correctness of the computing system. To make the above-mentioned approaches reasonable to use for verifying a distributed computing system with multiple component layers, some analog of the LVT approach would still need to be defined to compose multiple layers by verifying that each lower layer implements the higher layer.

Similar to above-mentioned research, LVT is based on the traditional state transition model. However, LVT is specifically oriented towards the verification of distributed computing systems with multiple layers. It defines how to structure the specifications of different layers so that the verifications of implementation correctness and their composition can be performed easily. With LVT, each layer in a distributed computing system is treated as a distributed programming language. For a relatively higher-level language, the purpose of verifying its implementation in terms of its lower-level language is to ensure that the lower-level representation of the higher-level statement makes transitions that are semantically faithful to those of the higher-level language. With the composition of all the verified layers, the semantics of the high-level distributed programming language at the top layer of the distributed computing system can be guaranteed with respect to the entire computing system. Therefore, at each given layer, state transitions involved are only constrained by the formal specification of the language semantics, rather than by the specification of desired behaviors or properties of any program or system written and developed in the language. In addition, the state transition model used in LVT is of a linearly but nondeterministically ordered interleaving semantics.

2.3 HOL

Machine-checked proof construction in theorem proving systems is generally accepted as being secure because the systems have attained the level of rigor and reliance upon established mathematical techniques. Representative theorem proving systems include the supporting proof system for Boyer-Moore logic (Boyer and Moore, 1988), HOL (Gordon and Melham, 1993), and PVS (Owre, Rushby, and Shankar, 1992).

LVT is independent of the theorem proving system used in the verification. However, to show the feasibility of LVT, it is necessary to apply LVT to the verification of a distributed computing system by using a particular theorem proving system. HOL is a reasonable choice to show the feasibility of LVT for two reasons. First, the authors have previous experience using HOL for verification. Second, LVT grew out of a previous project (Zhang, Shaw, Heckman, Benson, Archer, Levitt, and Olsson, 1994) on system verification using HOL, in which the composition of proofs of different layers proved difficult. LVT makes the difficult task more tractable.

HOL is a mechanized reasoning system that supports interactive theorem proving in a subset of higher order logic formulated by Alonzo Church. The system is implemented in the strongly typed functional language ML (Ullman, 1994) and retains the soundness characteristic from its predecessor LCF (Paulson, 1987). Over the years, HOL has been increasingly used in the formal verification of hardware systems and software systems. The following two features are particularly useful in the work presented.

First, as a strongly typed logic, HOL has basically three kinds of types. An atomic type denotes a HOL system built-in primitive type, e.g., *int* and *bool*. A compound type builds a new type from originally available type(s), e.g., *list* and *cartesian product*. Compound types can also be recursively defined by using data constructors. A function type builds a new type from two available types, one as the domain type and the other as the range type. For this research, the type mechanism in HOL facilitates the definition of abstract and recursive syntactic structures, the syntax-directed recursive semantic specification, and the use of structural induction in the proof involved.

Second, HOL allows the definition of embedded theories, such as the "programming language" at each layer, and is very expressive for formal definitions. As a higher order logic, it allows variables, arguments of functions, and results of function applications to range over functions and predicates (boolean valued functions). In addition, it allows the definition of curry-formed functions, a good introduction of which can be found in (Meyer, 1990). For any function $f\ (x,\ y)$ of signature $X \times\ Y \rightarrow Z$, its curry-formed function is $f\ x\ y$ of signature $X \rightarrow\ (Y \rightarrow Z)$. The arguments of f can be partially instantiated with left-association in function applications, e.g., $f\ (a{:}X)$ results in a function of signature $Y \rightarrow Z$. This provides the flexibility and expressiveness in formal specification. When a function has more than one argument, the same function definition is allowed to be used or applied in various ways to generate or denote different values including functions. HOL's expressiveness is shown below by a small example on the meaning definition for *Assignment* statement in a small sequential programming language. This definition can certainly be done without using the features of higher order logic and the curry-formed function definitions, but the definition will not be as terse as the definition below.

$\vdash$ *def*

```
meaning_assign(v:Variable) (e:Expression) (state:Variable→ Value) =
                change_state (v) (meaning_expression (e) (state)) (state)
⊢ def
change_state(v:Variable) (data:Value) (oldstate:Variable→ Value) (x:Variable) =
             IF (x = v) THEN data ELSE (oldstate x)
```

The two functions involved above are all curry-formed higher-order functions. Since a program state for sequential programs can be viewed as a set of bindings between program variables and their values, it can be defined in HOL as a function of signature (type) *Variable → Value*. For the *Assignment* statement, a function called *meaning_assign* is defined that is of signature *Variable → Expression → (Variable→ Value) → (Variable→ Value)*. A function is used as one of the arguments of *meaning_assign*. The result of applying *meaning_assign* is a new program state, a function as well. The function *change_state* is of signature *Variable → Expression → (Variable→ Value) → Variable→ Value*. When this curry-formed function is used in *meaning_assign*, only its left three arguments are instantiated so that the result of the function application is a function of type *Variable→ Value*. In the definition above, assume *meaning_expression (e) (state)* generates the value of expression e at the program state *state*.

HOL provides generic functions on lists. Later sections will use the *HD*, *TL*, *LENGTH*, *APPEND*, and *EL* functions. *HD* returns the head — i.e., the first element — of a given list. *TL* returns the tail — i.e., the list containing all but the first element — of a given list. *LENGTH* returns the length of a given list. *APPEND* returns a new list formed by appending its two arguments, which should be lists of the same type. *EL* (HOL's list element selector) returns the *ith* element of list l, where integer i and list l are arguments to *EL*.

3 The Technique

3.1 The State Transition Model

3.1.1 For Any Given Layer

All *primitive* statements at any given layer are atomic but with different granularities at different layers. 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 because the intermediate points of its execution are not observable to other processes. Thus, if two primitive statements, say A and B, are executed concurrently in two processes, the concurrent execution is modeled as either a state transition of A followed by that of B, or a state transition of B followed by that of A. Although the concurrent execution of two statements by two processes is modeled 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 execution 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., *guarded if* statement) is actually an interleaving of the execution of its atomic components and the execution of atomic primitives of other processes.

This issue of execution eligibility in the semantics of distributed languages does not have its counterpart in the semantics of sequential programming languages because the latter have no inter-process communication mechanisms. In any given state, some statements are always selectable and some are only conditionally selectable. 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) ...

Therefore, under the state transition model described above, the following three issues must be addressed in the semantic specification for a distributed language: the intra-process continuation that defines the dynamic decomposition of a composite statement into atomic transition steps; the intra-process sequencing of statements that defines the state transitions of two adjacent intra-process statements with potential interleaving of steps from other processes; and the execution eligibility for the system-wide sequencing that defines the allowable interleavings, i.e., the synchronization of concurrent execution of multiple processes. All these issues are addressed by LVT's semantic specification framework discussed in Section 3.2.

3.1.2 For Any Two Adjacent Layers

Since each layer in a distributed computing system is viewed as a distributed programming language, two layers are called adjacent layers when the relatively higher-level language is implemented by the lower-level language that provides services in support of functionality of the higher layer. As described below, LVT has given careful attention to what will happen between the adjacent language systems. 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 in Section 3.1.1, 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 microSR specification. For illustrative purposes, Figure 1 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",

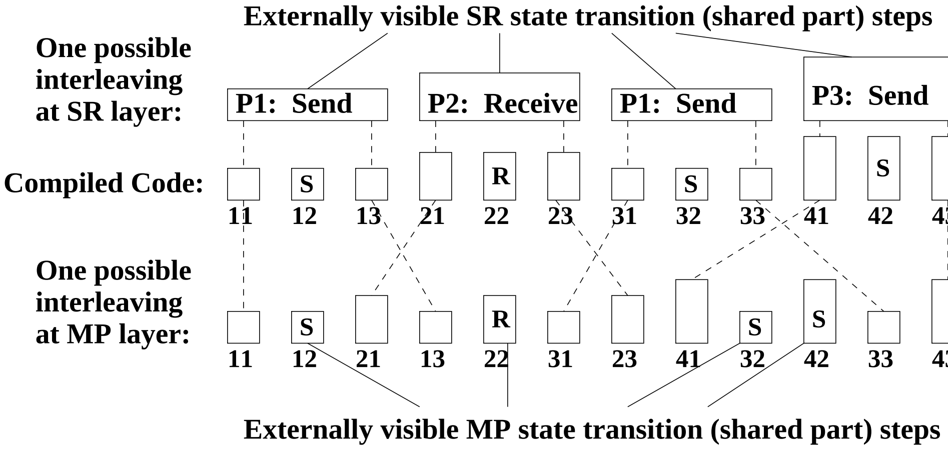

Figure 1: Interleavings at microSR and MP layers.

which corresponds to the "clean-up" after the instruction S. This overlapping is allowed at the MP layer because of the finer atomicity and interleavings. However, the instruction R can be selectable only when at least one message has been sent to the message queue by the instruction S.

Therefore, in order to verify the correctness of a distributed programming language implementation in terms of a lower-level distributed programming language, in addition to the formal semantic specifications for the two adjacent language layers, two issues must be addressed: the formalization of the language implementation relationship; and the formalization of a proof obligation for establishing the desired language implementation correctness. As discussed in Section 3.3, LVT formalizes a language implementation by the mappings of states, instructions, and processes between a given pair of language semantic specifications. LVT's proof obligation for a language implementation correctness is built upon the the equivalence of interleavings at two layers.

3.2 The Semantic Specification Framework

As discussed in Section 3.1, the semantic specification for a distributed programming language is very different from its counterpart for sequential languages. It must account for all possible interleavings and the effects of nondeterminism.

LVT provides a specification framework to address the generic semantic issues in distributed programming languages. It defines *Syntax, State, Continuation, Selection, Mean-*

ing, and *Co_Meaning.* These entities apply at any given layer in a distributed computing system. The following is an overview of these entities:

- *Syntax* represents the syntax of the language.

- *State* represents the execution state of a program of multiple processes.

- *Continuation* represents the "rest" of the computation that still has to be executed by a process in a given state. This relation formalizes intra-process syntactic continuation.

- *Selection* represents whether a statement is eligible to execute in a given state. This relation, therefore, reflects how processes synchronize. It defines the overall synchronization of multiple processes, formalizing the system-wide sequencing of valid interleavings.

- *Meaning* represents the meaning of statements executed by a single process. This relation reflects what changes can occur to the state due to the execution of a selectable statement.

- *Co_Meaning* represents the meaning of an entire concurrent program. This relation specifies the effect on the initial program state of executing a set of processes.

By using this framework, the semantic specifications of all three layers in the distributed computing system introduced in Section 1 have similar structures. However, the above required definitions differ in details at each layer, depending on how abstract the language being specified is from the computer hardware. As indicated in Section 3.1.2, the lower the abstraction level of a language, the smaller is its atomicity granularity. A lower abstraction level of a layer makes it simpler, more straightforward, and easier to understand the layer's semantic specification.

To illustrate how *Syntax, State, Continuation, Selection, Meaning,* and *Co_Meaning* should be defined, the rest of this section provides representative HOL definitions for these entities for the microSR language specification. These definitions use the generic HOL functions (e.g., APPEND) described in Section 2.3. To be more concrete, the discussion will use the example microSR program in Figure 2; see Section 3.2.1 for syntactic details. This

```
Co x:=3; y:= 5; Send maxq(x); Send maxq(y); Receive ansq(m)
//  Receive maxq(a); Receive maxq(b);
    If a>=b → r:=a [] a <= b → r:=b fi;
    Send ansq(r)
oc
```

Figure 2: Program MAXP: find the maximum of two numbers using message passing.

program contains two processes, P1 and P2, whose statement lists are separated by the // delimiter; variables that appear in each process are local to the process. P1 sends two numbers to P2; P2 computes the maximum of those numbers and sends the result back to P1. P1 uses two *Send*s to pass the two numbers to P2 because, for simplicity, microSR allows messages to contain only one parameter. Although this program is very simple, it serves as a good base for examples illustrating the entities of the LVT semantic framework.

3.2.1 Syntax

For the microSR language, *Syntax* is essentially the BNF for microSR. It is recursively defined, in part, as follows.

```
⊢ def Stmt ::= var := iexp
             | If bexp1 → stmt1 [] bexp2 → stmt2 fi
             | stmt1 ; stmt2
             | Send mq(iexp)
             | Receive mq(var)
             ...
⊢ def Prog ::= Co stmt1 // ... // stmtn oc
```

In the above, *iexp* stands for an integer expression, *bexp* stands for a boolean expression, and *stmt* stands for a statement or, due to the recursive definition, a list of statements.

The *If* statement is similar to Dijkstra's *guarded if* (Dijkstra, 1975 and 1976). It is non-deterministic. If both conditions are true, only one branch is chosen non-deterministically to execute. The [] delimiter separates the two branches.

The *Co* (Co-begin) statement specifies an entire program, which consists of a set of concurrently executed processes; process i executes the i-th arm (*stmti*) of the *Co*. The processes communicate via the message passing statements *Send* and *Receive*. Each of these statements specifies a *mq*, which stands for a *fifo* message queue (communication channel) shared by processes.

In some subsequent definitions, a process is represented by a process identifier *Proc_id*. These process identifiers are internal to the semantic specifications.

3.2.2 State

The semantic domain *State* represents the state of the entire program. It includes bindings for variables local to each process, the "program counter" for each process, and the contents of the (global) message queues. For example, during one possible execution of MAXP (Figure 2), the state would indicate that P1's x and y contain 3 and 5, respectively; P1 is about to execute its *Receive*; P2's a contains 3, but b and r have not been assigned; P2 is about to execute its second *Receive*; *maxq* consists solely of a message containing 5; and *ansq* is empty.

Henceforth, the bindings for variables local to a given process is referred to as the process's *Proc_local_state* and the contents of the message queues is referred to as the *Pool_state*. The "program counter" for each process is referred to as a *Thread*, as discussed next.

3.2.3 Continuation

A *Thread* represents what a given process needs to execute. It is a syntactic continuation, i.e., a representation of the code that the process still needs to execute. A *Thread* is a list of statements, the head of which is the statement to be executed next and the tail of which is the list of statements to be executed subsequently. A *Thread* changes as a process executes each statement. Its head is always "popped off". In some cases (e.g., for *Assignment* or *Send*), the new thread is just the tail of the current thread; in other cases (e.g., for *If*), the new thread will be a new list that includes the old thread. As an informal example based on MAXP (Figure 2), when P1 is just about to execute its first *Send*, its *Thread* is

Send maxq(x); **Send** maxq(y); **Receive** ansq(m)

After executing that first *Send* statement, its *Thread* is

Send maxq(y); **Receive** ansq(m)

When P2 is just about to execute its *If*, its *Thread* is

If a>=b $\rightarrow$ r:=a [] $a <= b$ $\rightarrow$ r:=b **fi**; **Send** ansq(r)

After executing that statement, its *Thread* is

r:=b; **Send** ansq(r)

Here, the new *Thread* consists of the statements in the true arm of the *If* appended with the tail of the previous *Thread*.

More formally, continuations are represented by the relation *Continuation*. It has the signature *Thread* $\rightarrow$ *Thread* $\rightarrow$ *Stmt* $\rightarrow$ *Proc_local_state* $\rightarrow$ *Bool*. *Continuation* takes as arguments, for a given process, the current *Thread*, a candidate new *Thread*, the statement that is about to be executed, and the local state; it returns the boolean value representing whether execution of the current statement in the current state can yield the new *Thread*. *Continuation* requires the local state so that, for example, it can evaluate the boolean expressions in *If*s — their values affect continuations, as seen in the example above. *Continuation*'s signature is curried, for reasons outlined in Section 2.3 and to be consistent with all definitions in the semantic specification. It returns a boolean, rather than a new *Thread*, because *Continuation* is a relation, not a function: due to non-determinism, multiple new threads are possible. The key other relations — *Selection, Meaning,* and *Co_Meaning* — are expressed similarly for the above reasons.

The definition of *Continuation* for two representative statements — *Send* and *If* — is:

$\vdash$ *def*
Continuation oldthread newthread (**Send** mq(iexp)) (ls:Proc_local_state) =
        (newthread = (TL oldthread))
Continuation oldthread newthread (**If** bexp1 $\rightarrow$ stmt1 [] bexp2 $\rightarrow$ stmt2 **fi**) ls =
        IF $\neg$(Mbexp bexp1 ls)$\wedge$ $\neg$(Mbexp bexp2 ls)
        THEN (newthread = (TL oldthread))
        ELSE (Mbexp bexp1 ls $\wedge$
                Continuation(APPEND[stmt1](TL oldthread)) newthread stmt1 ls) $\vee$
               (Mbexp bexp2 ls $\wedge$
                Continuation(APPEND[stmt2](TL oldthread)) newthread stmt2 ls)

The definition for *Send* shows the result of "popping off" the current, primitive statement: the *newthread* is simply the tail of the *oldthread*. The definition for *If* shows how to decompose a composite statement recursively to generate a thread that represents the syntactic continuation. *Mbexp* is the meaning function for boolean expressions; it takes as arguments the expression and a state in which to evaluate it. When both boolean expressions in an *If* evaluate (in the current state) to false, the *If* is simply "popped off". When one of the boolean expressions evaluates to true, the statement list in the corresponding branch is appended with the tail of the *oldthread*. The first statement in that branch then becomes the statement to execute next. The disjunction ($\vee$) is used to reflect *If*'s nondeterminism. When both boolean expressions evaluate to true, one of the branches is chosen.

3.2.4 Selection

Selection represents whether a statement is eligible to execute in a given state. Some statements — e.g., *Assignment* and *Send* — are always eligible, regardless of state. Other statements are eligible in only some states. *Selection*, therefore, defines how processes synchronize. A *Receive*, for example, is eligible iff a message is pending for the named operation. As an informal example based on MAXP (Figure 2), P1's *Receive* is eligible only when *ansq* has a pending message, i.e., after P2 has executed its *Send*.

The relation *Selection* has signature *State* $\rightarrow$ *Stmt* $\rightarrow$ *Proc_id* $\rightarrow$ *Bool*. *Selection* essentially takes as arguments the entire state of the program and the statement that a given process *Proc_id* wants to execute; it returns the boolean value representing the eligibility of such an execution in the current state.

The definition of *Selection* for three representative statements — *Send*, *Receive*, and *Sequence of statements* — is:

$\vdash$ *def*
Selection (s:State) (**Send** mq(iexp)) (p:Proc_id) = T
Selection s (**Receive** mq(v)) p = (unreceived_msg mq (get_pool_state s) $\geq$ 1)
Selection s (stmt1 ; stmt2) p = Selection s stmt1 p

As noted above *Send* is always selectable (*T* represents true). "**Receive** mq(v)" is selectable only when the given message queue *mq* contains at least one unreceived message, reflecting synchronous ("blocking") message receiving. The definition for *Sequence of statements* shows how *Selection* is recursively defined for composite statements. It indicates that the eligibility of a list of statements, when the process is about to execute the first, is just that of the first statement in that list. It does not assert anything about the eligibility of subsequent statements in that list (i.e., *stmt2*); the eligibility of those statements arises as dictated by the syntactic continuation of the first statement, i.e., by the relation *Continuation*.

3.2.5 Meaning

Meaning specifies the complete effect on a given state of a process executing a (selectable) statement. As informal examples based on MAXP (Figure 2), P1's second *Assignment* changes *y*'s binding in P1's local state, P1's first *Send* inserts a message containing 3 into *maxq*'s message queue, and its second *Send* inserts a message containing 5.

Because *Meaning* also deals with composite statements, it defines the meaning of all statements executed by a process. Continuing the previous example, between the two *Send*s, P2 might execute its first *Receive*, thereby removing the message containing 3 from *maxq*'s message queue. Thus, the *Meaning* of the composition of the two *Send*s is not just the *Meaning* of one followed by the *Meaning* of the other. It must also take into account possible interleavings that occur between such primitive statements.

More formally, statement meanings are represented by the relation *Meaning*. It has the signature *Stmt→State→State→Proc_id→Bool*. The expression *Meaning statement s1 s2 p* is true if *s2* (the new state) can be reached from *s1* (the old state) by executing *statement* in process *p*. For a primitive statement, the definition of *Meaning* relies on the relation *m_atomic_stmt*. Recall that a primitive statement has an atomic state transition. For a composite (i.e., non-atomic) statement, the definition of *Meaning* depends on the *Meaning*s of its constituent statements and, as noted above, must account for valid interleavings.

The definition of *Meaning* for two representative statements — *Send* and *Sequence of statements* — is:

⊢ *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

The definition for *Send* is just that of *m_atomic_stmt*, described below. The definition for *Sequence of statements* shows how the state transition of this composite statement is decomposed into two state transitions that might be further decomposed. *Continuation* and

Selection are used to determine the continuation and eligibility of the composite statement. (The *get_thread* function maps a state and a process-id (*Proc_id*) to the process's continuation (*Thread*).) The relation *m_proc_Seq* is used to assert the possible existence and the validity of the state transition interleavings between the two state transitions *Meaning stmt1* and *Meaning stmt2*; its definition is given below.

The relation *m_atomic_stmt* shows the effect of a process executing a primitive statement. An atomic state transition is allowed only if the statement is next to execute according to *Continuation* and is eligible according to *Selection*. The definition of *m_atomic_stmt* for the representative statements *Assignment* and *Send* is:

```
⊢ 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
```

The above definitions relegate the work of defining the actual state changes to *m_Assign* and *m_Send*.

m_Assign shows the effect an *Assignment* has on a process local state. It is similar to *meaning_assign* in Section 2.3 in that the new state is the same as the old state except for the value (as determined by *Miexp*, the meaning function for integer expressions) of one process local variable. The one notable difference is that *m_Assign* also indicates that *Assignment* does not change the shared message queues. It does that below by asserting the equality of *get_pool_state* in the old and new states.

```
⊢ 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) (old_local_state:Proc_local_state)(x:Var) =
       IF (x = v) THEN data ELSE (old_local_state x)
```

m_Send shows the effect a *Send* has on the shared pool state, and that *Send* has no effect on the process local state. The specific effect is that a single message (created by function *mk_msg*) is added into the given message queue. More precisely, the relation *mq_add_msg* asserts that messages from the same sender are well ordered.

```
⊢ 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) (old_pool:Pool_state) (mq':msg_queue) =
  IF (mq' = mq) THEN new_mqvalue ELSE (old_pool mq')
```

To complete the definition of *Meaning*, the effects of possible interleavings of other processes, as discussed earlier in this section, need to be handled. The relation *m_proc_Seq* shows the effect on a state by an intra-process sequence of state transitions of one given process, possibly interleaved with system-wide valid state transitions associated with other processes. The relation *m_sys_interleaving* specifies the existence of possible interleavings between two intra-process state transitions. *m_proc_Seq* also asserts that effects of other possibly interleaved processes on the state will not change (1) the local state of this given process and (2) the effect of this process on the shared message pool, i.e., the messages that have been already sent and received by this process and the order in which these messages are sent and received. The definition of *m_proc_Seq* uses the function *get_local_state* as part of (1) and the function *proc_effect_on_pool* as part of (2).

```
⊢ def
m_proc_Seq (transition1,transition2: 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 (get_pool_state s3) p) = (proc_effect_on_pool (get_pool_state s4) p))∧
    transition1 s1 s3 p ∧ transition2 s4 s2 p
```

3.2.6 Co_Meaning

Co_Meaning represents the meaning of an entire microSR program, i.e., the overall effect of the concurrent executions of the processes of the program's *Co* statement. As an informal example based on MAXP (Figure 2), the interesting part of the overall effect is that *m*'s binding in P1's local state is 5; the other parts are the other bindings for local variables and that the message queues are empty.

Just as *Meaning* needs to take into account possible interleavings between primitive statements, *Co_Meaning* needs to take into account, for each process, the effect of possible executions of other processes between the start of the *Co* and the start of the given process and between the end of the given process and the end of the *Co*. For example, in the MAXP program, immediately after P2's *Send* (and P2 ends), *maxq*'s message queue contains a message, but *maxq*'s message queue is empty at the end of the entire *Co* due to P1 executing its *Receive*.

More formally, the meaning of a *Co* statement is specified mainly by the relation *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* arm of the *Co* statement. Its definition reflects the possible differences, as explained above, between the process's start state *s1'* and the entire program's start state *s1* and between the process's final state *s2'* and the entire program's final state *s2*. However, similar to the specification of *m_proc_Seq*, the effect of other processes on the state will change neither the local state of this process nor the effect of this process on the shared pool.

⊢ *def*
Program_Meaning (**Co** stmt_1 // ... // stmt_n **oc**) ([pid_1, ..., pid_n]) (s1:State) (s2:State) =
        Co_Meaning ([stmt_1, ..., stmt_n]) ([pid_1, ..., pid_n]) (s1:State) (s2:State)
⊢ *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))

3.3 Language Implementation Correctness

As briefly mentioned at the end of Section 3.1.2, to prove the correctness of an implementation of a higher-level distributed language in terms of a lower-level distributed language in a distributed computing system, the following must be formalized.

- The semantic specifications for each of the two adjacent language layers, as described in Section 3.2;
- The language implementation relationship between the two language layers;
- The proof obligation for establishing the desired language implementation correctness. This proof obligation provides the specification of language implementation correctness that must be proved as a theorem.

3.3.1 Language Implementation Relationship

As shown in Figure 3, no matter how a distributed computing system is hierarchically decomposed, it is necessary to specify three mappings between any two adjacent language layers to fully represent the language implementation relationship. A *State* represents a program state at a given layer in the distributed computing system. An *Action* stands for the action resulting in a state transition. At the programming language layer, it is a statement of the language; at the multiprocessor layer, it is a machine instruction or a system call. Similarly, an *Actor* stands for the unit performing the state transition. At the language layer, it is a process; at the multiprocessor layer, it is a processor; at the network interface layer, it is a network unit. So *MapDownAction* and *MapDownActor* map more abstract *Action, Actor* at a higher layer into their more concrete correspondences at a lower layer. For instance, *MapDownAction* is the compiler between a programming language layer and a multiple processor layer, whereas it is the correspondence of machine instruction and its hardware steps between a processor layer and a network interface layer. For each state at a higher layer, there are multiple corresponding equivalent states at a lower layer because of the finer state transition steps and interleavings. The mapping *MapUpState* provides the state abstraction between two adjacent layers. The generic signature of these three mappings are shown below.

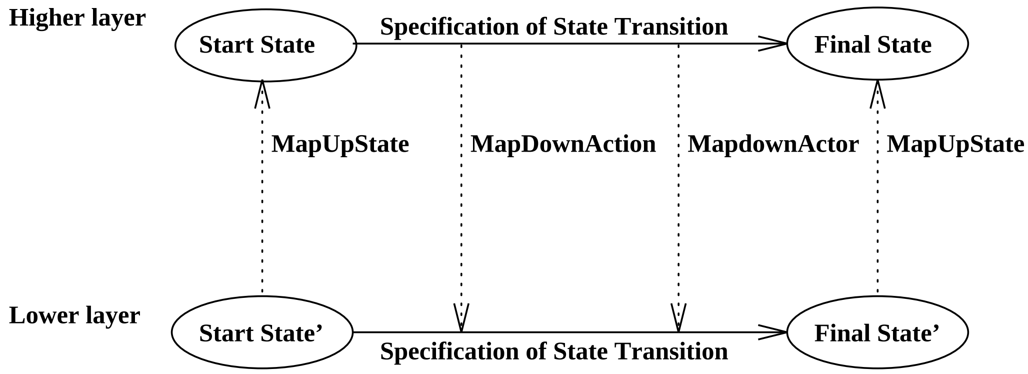

Figure 3: Necessary mappings between layers

MapDownAction: high_Stmt $\rightarrow$ low_Stmt
MapUpState: low_State $\rightarrow$ high_State
MapDownActor: high_Proc_id $\rightarrow$ low_Proc_id

3.3.2 The Proof Obligation for Language Implementation

Figure 4 shows a schematic of the specification of the language implementation correctness for two adjacent distributed language layers, where the higher-level distributed language is implemented in terms of a lower-level distributed language. The correctness of the higher-level distributed language implementation has to be verified with respect to the pair of language semantics, as well as to the formalization of the language implementation relationship, the three mappings between the two language layers. The "execution" of the "generated" lower-level instructions, with respect to the start and final states at the lower layer, 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. As specified below, the relation *Stmt_implemented_correct* for the language implementation correctness is built upon the relation *Equivalent_interleaving* for the equivalence of interleavings at two adjacent layers. The relation *Stmt_implemented_correct* is also called the proof obligation because it is required to be proved as a theorem to establish the language implementation correctness. Notice that, in proving the language implementation correctness, the premise of the proof obligation has to be proved as a theorem first.

Because of the nondeterminism at two adjacent layers and the finer atomicity and interleavings at the lower layer, for a state transition or a sequence of state transitions, the

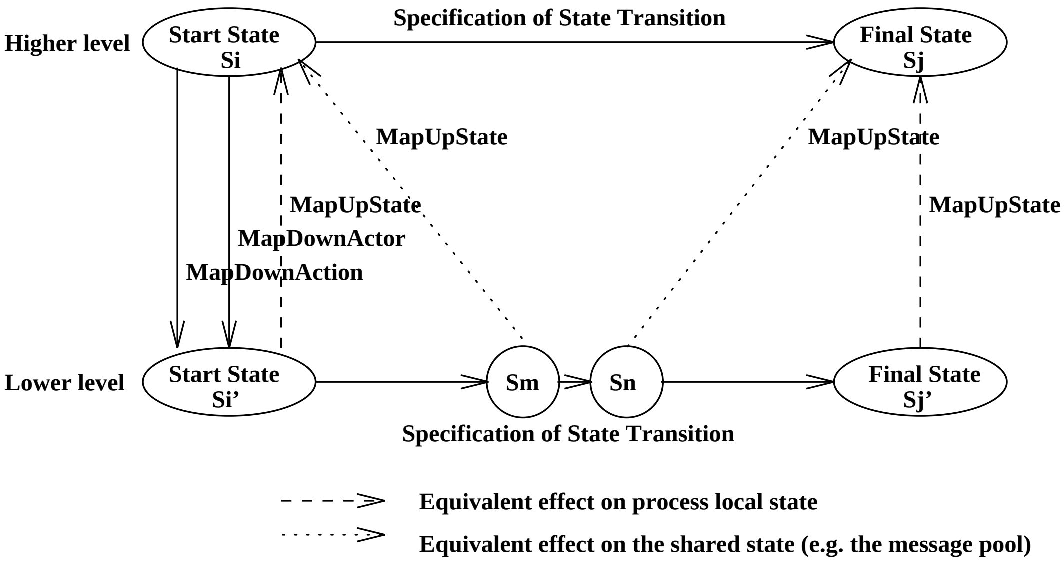

Figure 4: The correct implementation relationship between two adjacent layers.

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 definition of the proof obligation. As shown in Figure 4 and in the specification of the proof obligation, a correct language implementation requires that the two layers have equivalent interleavings and the changes to the states must satisfy the language semantic specification.

- The equivalence of effects on the local states of processes is defined with respect to start and final states at both layers, as well as to a given process at the higher layer. A correct language implementation must guarantee that the local state of the start state and the local state of the final state at the higher layer, with respect to a particular process executing a statement, are respectively identical to the local states abstracted from the corresponding start and final states at the lower level. As required by the relation *high_effect_on_local*, the change to the given process local state must satisfy the meaning of the statement executed.

- The equivalence of effects on the shared states (e.g., message pool states) is defined with respect to a given process at the higher layer, to the start and final states at the higher layer, as well as to two intermediate states Sm and Sn at the lower layer. A correct language implementation must guarantee that the effect of state transition on pool states at the higher layer from start and final states, with respect to a particular process executing a statement, is identical to the effect of this process on the pool states abstracted from corresponding states Sm and Sn at the lower level. *high_proc_effect_on_pool* gives the contribution to a pool state from any given process, which includes messages sent to and received from the shared message pool by this process and the order in which those messages are sent and received. As required by the relation *high_effect_on_pool*, the change to the pools state contributed by the given process must satisfy the meaning of the statement executed.

The two intermediate states Sm and Sn actually 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.

⊢ *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_proc_effect_on_pool (high_get_pool_state si) p =
 high_proc_effect_on_pool (high_get_pool_state (MapUpState sm)) p) ∧
(high_proc_effect_on_pool (high_get_pool_state sj) p =
 high_proc_effect_on_pool (high_get_pool_state (MapUpState sn)) p)
⇒
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_state si) (high_get_pool_state sj) p

4 Verification of the 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.

4.1 MP Machine Specification

In accordance with LVT's overall semantic specification framework, the semantics of the instruction execution at the MP layer is formalized by generating the definitions of Section 3.2. Following the framework 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 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 elements) set of basic instructions like ADD, JMP (jump to), LD (load), and STO (store) —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 PC 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 (jump on zero) instructions are the only exceptions to this rule because they modify the program counter. Since all the VM instructions are common to sequential computations and since the SEND system call is an asynchronous one, the execution of any of them does not need to be blocked at any time in any state. Therefore, all of the VM instructions and the SEND system call satisfy the *Selection* relation in any given state. 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 there are multiple processors executing code in a nondeterministic order at the MP layer, it must be allowed for instructions from multiple processors to interleave their execution. Thus, as explained in Section 3.2, the effects of both intra-process and inter-process sequencing on the MP state must be specified. 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.

4.2 Mappings Between the MP Layer and the microSR Layer

As mentioned in Section 3.3, 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 a function on the process IDs.

Mapping down the code is more complicated. For this, a compiler for microSR is formalized and 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

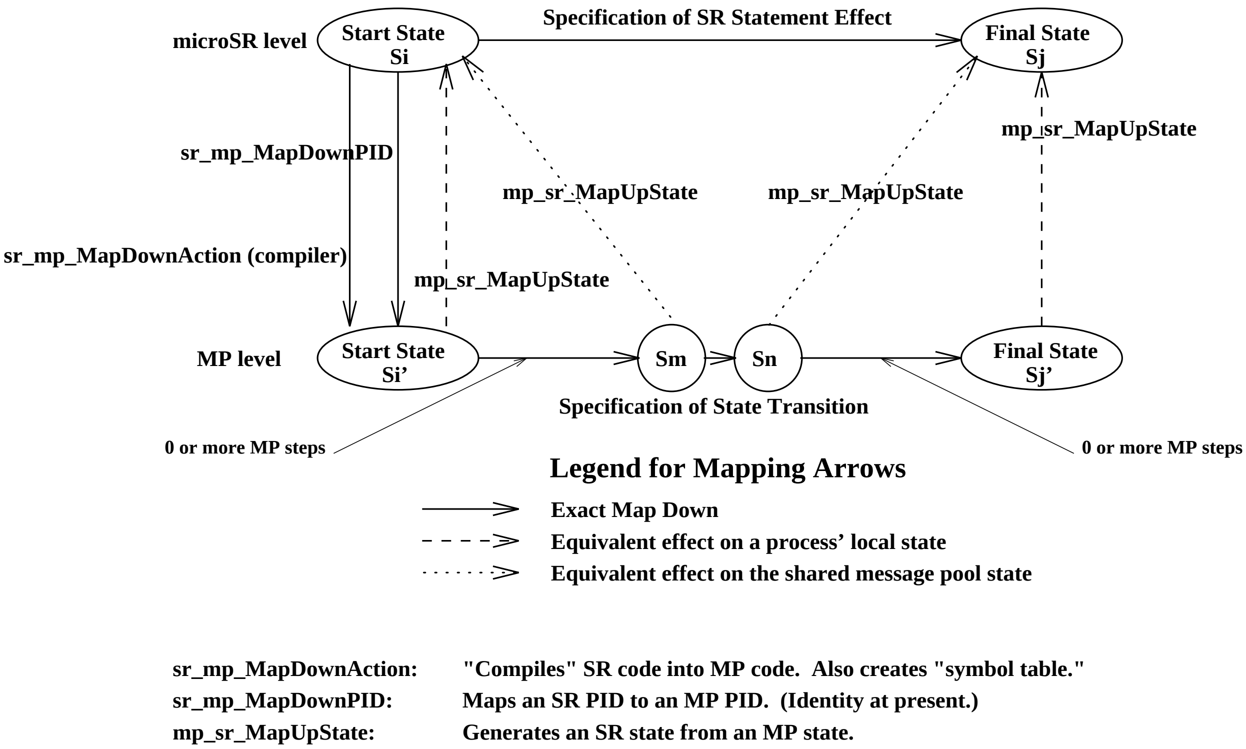

Figure 5: A Verified microSR Implementation

processors. Therefore, as indicated previously, the execution of MP code can be interleaved with other code executed by other processors. For the initial compiler, little attempt at optimization has been attempted.

4.3 Implementation Correctness

Figure 5 illustrates the proof obligation of the implementation of microSR, which is a concrete instantiation of the generic proof obligation shown by Figure 4. 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)
⇒
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_proc_effect_on_pool (SR_get_pool_state si) p =
 SR_proc_effect_on_pool (mp_sr_MapUpPool sm) p)∧
(SR_proc_effect_on_pool (SR_get_pool_state sj) p =
 SR_proc_effect_on_pool (mp_sr_MapUpPool sn) p)
⇒
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
```

5 Verification of the MP Layer System Calls

5.1 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 in 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 5.2.

As shown in Figure 6, MP SEND and RCV are both implemented by a sequence of three network level statements (operations): an initialization operation Init, a transfer operation Send or Receive, and a termination operation Term, which completes the system call. In the case of the SEND system call, the transfer 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 3.2. 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 3.2, 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 yet transferred to the MP machine.

5.2 The Mappings between the Network Layer and the MP Layer

The implementation of MP system calls is formalized by 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

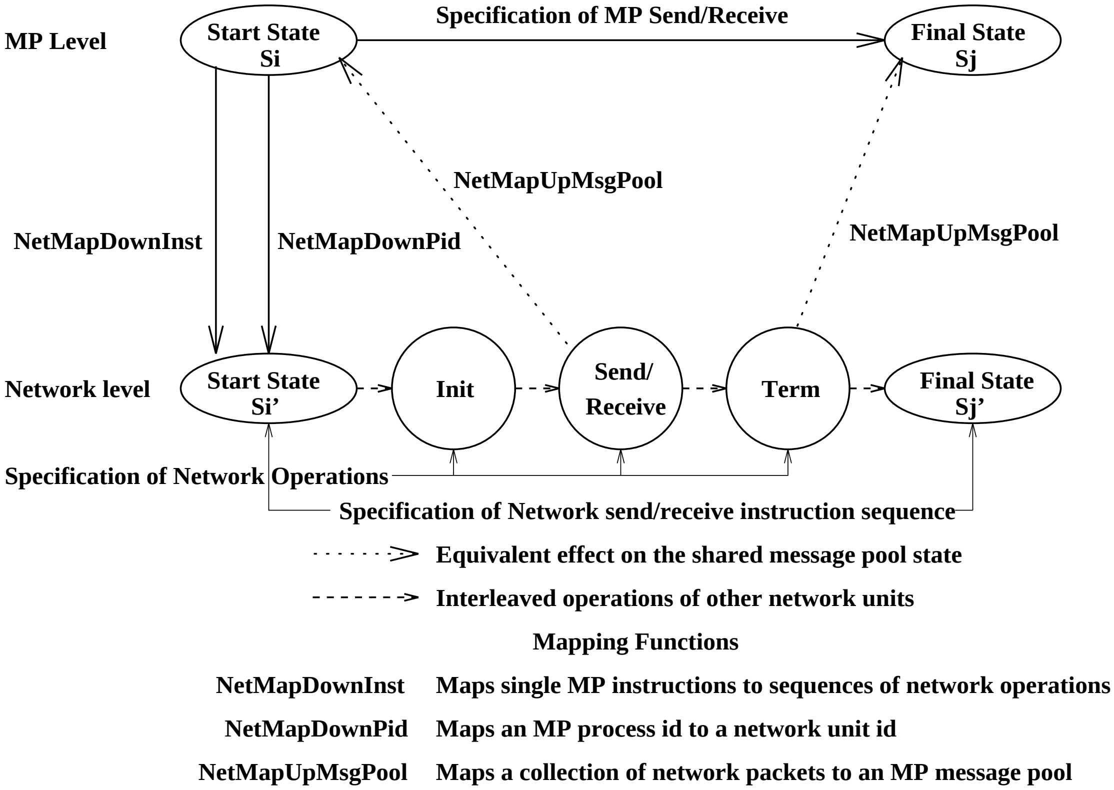

Figure 6: A verified MP system call implementation.

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.

5.3 Implementation Correctness

A diagram of the proof obligation for the MP layer implementation is shown in Figure 6. It is also a concrete instantiation of the generic proof obligation shown by 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_proc_effect_on_pool (MP_get_pool_state si) p =
 MP_proc_effect_on_pool (NetMapUpMsgPool sm) p) ∧
(MP_proc_effect_on_pool (MP_get_pool_state sj) P =
 MP_proc_effect_on_pool (NetMapUpMsgPool sn) p)
⇒
MP_effect_on_pool stmt (MP_get_pool_state si) (MP_get_pool_state sj) p

6 Conclusion

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. LVT makes this task more tractable by providing a generic semantic specification framework and a generic proof obligation. This technique has been applied to the verification of the microSR implementation through layered proofs of a distributed computing system with three layers. This work demonstrates that the implementation correctness of a distributed language with respect to the entire distributed computing system is verifiable through this *layered verification technique*. The layered verification of a distributed computing system guarantees the correctness of the compiled code, most notably the inter-process communication, that makes reference to operating system calls, the correctness of the operating system calls in terms of network calls, and the correctness of network calls in terms of network transmission steps.

The current distributed computing system in this work is small, but non-trivial, resulting in non-trivial layered proofs in HOL. This research is currently directed towards evolving this small layered distributed computing 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 and more advanced communication mechanisms. However, any changes and extensions usually require a reverification of correctness goals and the work involved is certainly significant. Therefore, an evolutionary approach to verification that reflects these changes and extensions to the distributed computing system rather than a complete reverification is highly desirable and currently under investigation. It is intended to demonstrate in the future that the layered verification of a computing system can evolve in unison with the evolution of the system design, and that the technique presented is applicable to the verification of larger distributed systems.

Acknowledgements

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

The anonymous reviewers and the editor provided many constructive suggestions and comments, which greatly helped to improve the presentation of this paper.

References

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

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

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

Boyer, R. S. and Moore, J. S. (1988), *A Computational Logic Handbook*, Academic Press, Boston.

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

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

Dijkstra, E. W. (1975), 'Guarded commands, nondeterminacy, and formal derivation of programs', *Communications of the ACM*, Vol.25, No.8, pp. 453-457.

Dijkstra, E. W. (1976), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J.

Francez, N. (1992), *Program Verification*, Addison-Wesley publishing Company Inc, England.

Good, D. I. and Young, W. D. (1991), 'Mathematical Methods for Digital Development', Technical Report 67, Computational Logic, Inc.

Good, D. I., Kaufmann, M., and Moore, J. S. (1992), 'The Role of Automated Reasoning in Integrated System Verification Environments', Technical Report 73, Computational Logic, Inc.

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

Joyce, J. J. (1989), '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, pp. 177-201.

Meyer, B. (1990), *Introduction to the Theory of Programming Languages*, Prentice Hall International (UK) Ltd.

Owre, S., Rushby, J. M., and Shankar, N. (1992), 'PVS: A Prototype Verification System', *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 607, pp. 748-752.

Paulson, L.C. (1987), *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge; New York : Cambridge University Press.

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

Ullman J. D. (1994), *Elements of ML Programming*, Prentice-Hall, Inc. Englewood, New Jersey.

Windley P. J. (1993), 'A Theory of Generic Interpreters', *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 683, pp. 122-134.

Zhang C., Shaw R., Heckman M. R., Benson G. D., Archer M., Levitt K. N., and Olsson R. A. (1994), 'Towards a Formal Verification of a Secure Distributed System and its Applications', Proceedings of the 17th National Computer Security Conference, Baltimore, pp. 103-113.