TOWARDS A FORMAL VERIFICATION OF A SECURE DISTRIBUTED SYSTEM AND ITS APPLICATIONS[1]

Cui Zhang, Rob Shaw, Mark R. Heckman, Gregory D. Benson, Myla Archer, Karl Levitt, and Ronald A. Olsson

Department of Computer Science
University of California, Davis, CA 95616
Email: Last-Name@cs.ucdavis.edu

Abstract

This paper presents research towards the formal specification and verification of a secure distributed system and secure application programs that run on it. We refer to the whole system — from hardware to application programs written in a concurrent programming language — as the Silo, and to a simplified view of the Silo as the miniSilo. Both miniSilo and Silo consist of a collection of microprocessors interconnected by a network, a distributed operating system and a compiler for a distributed programming language. Our goal is to verify the full Silo by mechanized layered formal proof using the higher order logic theorem proving system HOL. This paper describes our current results for verifying the miniSilo and our incremental approach for evolving the verification of the miniSilo into the verification of the full Silo. Scalability is addressed in part by extending the distributed operating system with additional servers which in turn provide services that extend the programming language.

Keywords: verification, distributed operating systems, security servers, distributed programming languages.

1 Introduction

This paper describes our research on a long term project called the Silo. This project is aimed at verifying a complete distributed computer system by mechanized layered formal proof. Our layered system includes a set of microprocessors, a network model, the operating system kernel and servers (some in support of security) running on each microprocessor (hence, a secure distributed operating system), the concurrent programming language microSR (a derivative of SR [1]), and a Hoare-like programming logic. Each layer will be formally modeled as an interpreter that interacts with the other layers. Our layered approach will allow us to verify that secure and distributed applications run correctly on the entire system. In its final form, the Silo will be somewhat limited when compared to "real" computer systems; however, we hope it will be the most comprehensive distributed computer system that is verified and demonstrates a methodology for "full system verification" of distributed systems.

The CLI stack [2] has shown the feasibility of full system verification for a sequential system using a layered proof technique, but their model does not allow for concurrency and distributed programming, nor have they fully integrated the operating system into their "stack". When we began specifying the Silo system, we realized that an incremental approach is necessary for revealing unforeseen difficulties and for making the formal proof more manageable. Rather than attempting to specify and prove the entire Silo, we have identified a subset of the Silo to specify and prove correct by limiting the scope of each layer to reduce the complexity. As shown in Figure 1, we refer to this simplified view of the Silo as the miniSilo. As our preliminary results on miniSilo have

[1]This work was sponsored by the National Security Agency University Research Program under contract DOD-MDA904-93-C-4088 and by ARPA under contract USN N00014-93-1-1322 with the Office of Naval Research.

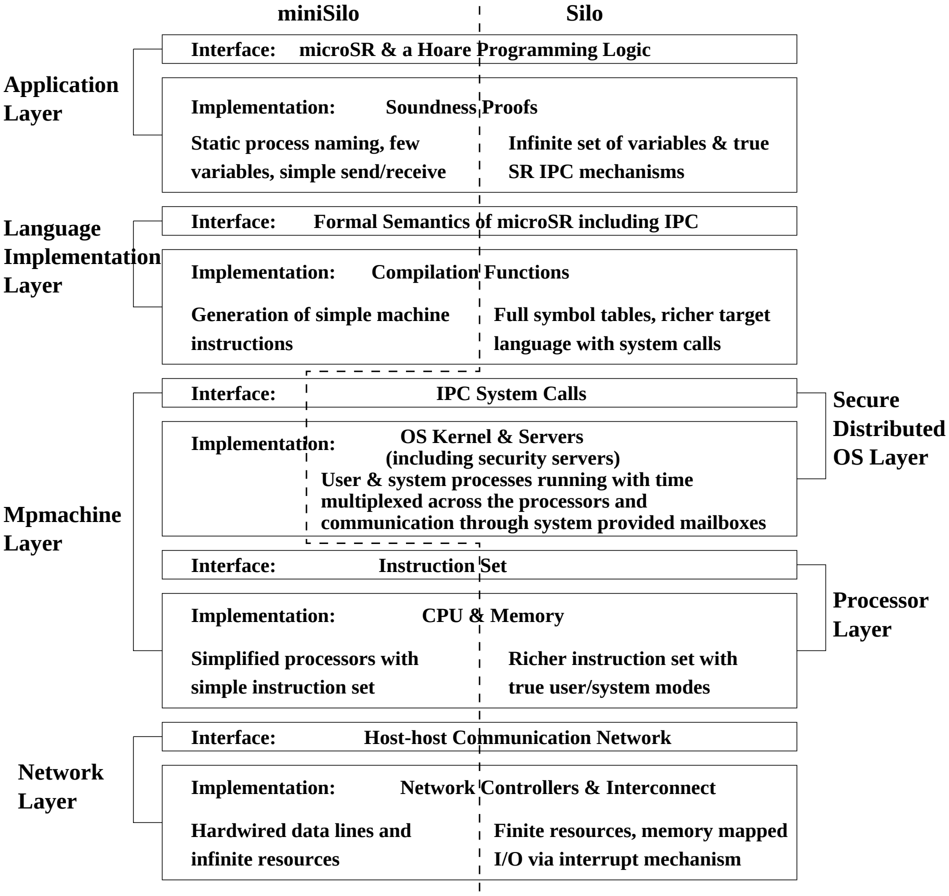

Figure 1: Overview of UCD Silo and MiniSilo

shown the usefulness of our layered proof methodology, we are now growing the miniSilo system into the full Silo by developing the system and proof by incrementally adding functionality to all layers. Our specification, verification, and augmentation process is being carried out using the Cambridge HOL theorem prover [8], because it allows the definition of embedded theories, such as we are using for a programming logic of concurrency and a generic model of a layer. We also hope our work demonstrates the expressiveness, flexibility, and feasibility of higher order logic in formal specification and verification for more complicated computer systems, including a concurrent programming language that support security applications and a distributed operating system.

This paper concentrates on our miniSilo effort, as a step in the full Silo effort. Section 2 describes our work on the network layer. Section 3 gives our work on the mpmachine layer. Section 4 describes our effort on the language implementation for microSR. Section 5 presents the Hoare logic derived from the microSR semantic specification. Section 6 concludes our work.

2 The Network

2.1 The Network for MiniSilo

The lowest layer of miniSilo consists of a network which allows individual processors (vmachines, see Section 3) to communicate through message passing. The miniSilo network consists of a set of processors and an interconnect service. Each processor communicates with the network through a

Network Interface Unit (NIU), as shown in Figure 2. In miniSilo we assume that a processor has dedicated, hardwired data lines that interface directly with an NIU. The network provides reliable transmission of messages and preserves message ordering between communicating processors.

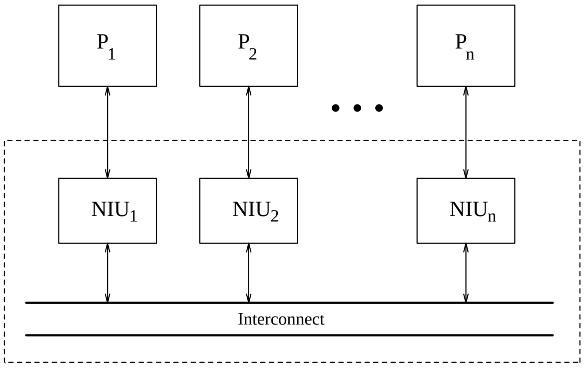

Figure 2: The MiniSilo Network

The miniSilo network is specified abstractly. By specifying the network in general terms, we do not impose any restrictions on the network topology or on the communication protocol. We do ensure that the network provides the properties that the higher miniSilo layers assume of the network. Later, if desired, one could develop an implementation of the abstract network specification. The next logical layer is the protocol layer. There has been considerable work on the verification of network protocols [5, 12], which could be used to implement the abstract specification presented. For "complete" verification the protocol layer must ultimately be specified in terms of the underlying hardware. Protocol and network hardware verification are beyond the scope of this project. The network is also specified operationally, where each NIU is modeled as an interpreter that reads and modifies state. The entire network is modeled as the composition of all the NIUs. The network interpreter is driven by send requests from the processors. Send requests result in receive requests from an NIU to a processor, which allows for nonblocking I/O at the operating system level. Sends and receives are accomplished through memory mapped I/O.

The set of NIUs are modeled as a fully connected network through send and receive queues, collectively called in-transit queues. For n processors, each NIU has $n-1$ send queues and $n-1$ receive queues. Each queue is shared by exactly two NIUs, one NIU views the queue as a send queue and the other NIU views it as a receive queue. The send and receive queues form the *InTransit_State*. The *NIU_State* for each NIU combined with the *Intransit_State* form the *Network_State*. The specification of the network interpreter is a relation, *Network_State→Network_State→Bool*. This interpreter is used to prove properties about the network itself, as well as to serve as an implementation for the higher layers of miniSilo.

Because the miniSilo network specification is given in terms of abstract operational semantics, we need to prove certain safety properties to ensure that the network functions correctly. The most important safety property is the ordering of messages between communicating processors. This property follows from the representation of the *InTransit_State*. Other safety properties, such as no duplication of messages, are also verified.

2.2 The Network for Silo

The proof obligation of the mpmachine requires us to verify that the network specification combined with the vmachine specification logically imply the mpmachine specification. In miniSilo, the distinction between the mpmachine communication abstractions and the network abstractions are small, but this will change once the full Silo is developed and each layer is expanded to more realistic specifications of a distributed system. In particular, the network specification will be modified in two respects. First, the network will be specified in terms of finite resources rather than infinite resources. Currently the specification allows infinitely many messages to be present in the in-transit queues. Therefore, each NIU is always ready to send another message, the processor is not required to wait or resend messages. Moving from infinite queues to finite queues entails certain specified error conditions and can result in storage channels. In miniSilo, we also assume that the message being transferred is a single, but infinite integer. We intend to alter the specification to handle finite packets. Second, the interface between an NIU and a vmachine will be enhanced to one based on memory-mapped I/O and interrupts rather than memory-mapped I/0 alone. This will allow the operating system to implement non-blocking I/O, and more importantly, allow for more than one process and operation per processor as described in Section 3. The new processor to NIU interface will also be enhanced to handle simple error conditions such as a network busy error or packet lost error, both of which will result in the processor resending the packet. Again, there are security implications to these decisions, which we will consider.

3 The Mpmachine

3.1 The MiniSilo Mpmachine

The miniSilo abstraction mpmachine represents multiple processors, each running a single process. Processes communicate by passing messages through a network. From a user process's point of view, the operating system interface appears as an "extended machine", consisting of the basic machine instructions plus communication primitives (system calls). The communication primitives are used to send and receive messages, through message queues. MiniSilo has one message queue per process, where only one process can read from the queue and all other processes can send messages to the queue.

The vmachine specification describes a single processor in miniSilo. Each vmachine[2] consists of an infinite set of registers, an infinite set of memory locations, and a program counter. Since these are modeled in HOL using natural numbers, each location may hold a non-negative integer of any size. A single vmachine operates much as one would expect, interpreting a typical set of simple machine instructions consisting of load, store, arithmetic, comparison, and branching instructions. It can not, however, issue any kind of communication action with other vmachines; the mpmachine provides this ability. This modularization is intended to isolate the processor from changes in the network hardware — the mpmachine is responsible for the compatibility of these two lower components and for defining the pool of message queues and system calls. Neither component depends upon the other's specification in any way.

An mpmachine contains N vmachine processors and N network interface units (NIUs) connected to a bus. Within the mpmachine specification, however, this bus is abstracted as a pool of queues. This pool contains one queue for each NIU, representing the ordered list of pending messages destined for the vmachine corresponding to the particular NIU. The external appearance of an mpmachine, therefore, is an N-tuple of vmachines (whose appearance is "passed-up", unaltered), plus a pool of "in-transit" message queues.

Similarly, the language interpreted by an mpmachine is an N-tuple of lists of instructions. The set of instructions contains all the operations executable on a vmachine, plus communication primitives. Similar to earlier efforts [4, 10], the actual operation of the mpmachine is modelled with

[2]Initially, we chose this term as an abbreviation of "virtual machine". Presently, however, "vanilla machine" is perhaps more appropriate

transition relations. Each kind of transition allows a single component of the N-tuple to advance a single step. To issue a vmachine instruction, only the state of the corresponding vmachine hardware is affected. To issue a communication primitive, however, the global pool of queues may be altered as well.

The HOL specification of this machine model consists of straightforward type definitions for the objects described, plus the transition relation associated for each kind of mpmachine instruction. These relations have the type $Args \to MPprocess \to MPprocess \to Vid \to Bool$. The type Args characterizes the numerical operands to the instruction. An MPprocess represents a pair whose first component is the local state of the vmachine which is executing this instruction, and whose second component is the pool of queues. Finally, Vid is the index of the executing vmachine; this information is not available within an MPprocess. If we were to include, say, a read-only "processor id register" in each vmachine, then the information in Vid above would become redundant.

From this type definition, we see that the following question can be answered of each mpmachine instruction: Given the indicated operands, and the indicated initial configuration of the mpmachine, is it possible to arrive in a given configuration after the indicated processor executes this instruction? For example, the relation for a simple vmachine jump instruction would require that the pools in both MPprocess objects are identical, because a jump does not affect communications. Moreover, the underlying vmachine specification would ensures that the register and memory contents of the vmachine object within the first MPprocess must also be identical to the corresponding vmachine within the second MPprocess. Only the processor's program counter will differ between the two configurations, and this difference must agree with the target location given in the operand to the jump (indicated in the Args). The mpmachine specification does not directly contain these facts, but rather defers to the vmachine specification itself. As an example of message passing, if the instruction in question were a receive operation, both the processor and the pool contents will differ accordingly. In particular, after the instruction is complete, the destination register in the processor will contain the received value, and the appropriate queue in the pool (indicated by the Vid) will have one less message than it did before the instruction began.

Armed with a semantic relation for each instruction, the mpmachine specification only requires two more definitions to encompass the complete system behavior. The first of these, is an inductive definition of how a thread, an instance of a sequential program piece, may legally execute for $k \geq 0$ steps. To execute for zero steps, both the initial and the final MPprocess must be completely identical. To execute for $k > 0$ steps, there must exist an intervening MPprocess value, call it M, such that the appropriate semantic relation allows a one-step transition from the initial state into M, and the final $(k-1)$-step transition from M into the final state is allowed inductively. The second definition describes the legal behaviors of complete programs on an mpmachine, and it is not inductive. Here, a final state of the entire system is reachable from an initial one precisely when the corresponding initial and final MPprocess's for each component of the program are allowed by the above inductive definition, for some $k \geq 0$.

3.2 Growth to Complete Silo

The complete Silo system consists of multiple processors, connected by a network and each running a copy of the Silo operating system. The operating system design is based on the kernel and server model used, for example, in Mach [14] and in Synergy [15]. The kernel provides a multi-programmed, message passing environment for the server processes and user processes on a particular processor. The abstraction of a distributed system is maintained by the servers. As shown in Figure 3, from a user process's point of view, the operating system interface in Silo will extend that of miniSilo with richer basic machine instructions and system calls. In this way, the

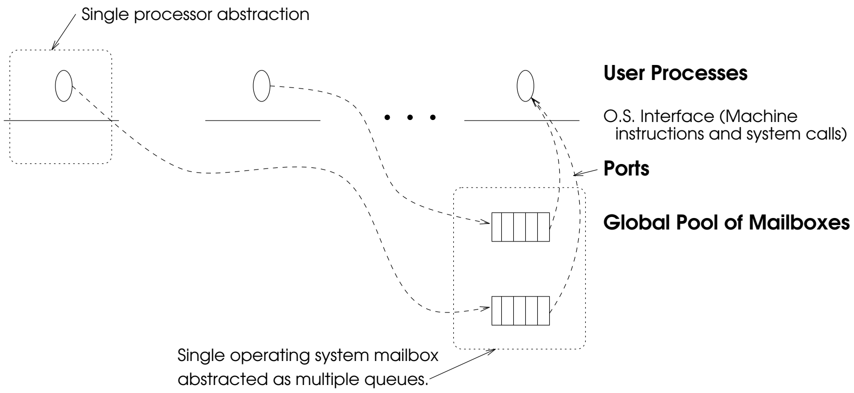

Figure 3: Operating System Specification: View from User Process

language work can proceed concurrently with the operating system work. Silo includes additional system calls for processes to create message queues, called *mailboxes*, and for processes to request access to specific mailboxes. The mailbox management calls are subject to a system security policy implemented by a security server, as shown in Figure 4. These calls, while an essential part of the Silo system specification, are only relevant to user processes when an application is initially loaded and, therefore, do not require significant changes to our language work.

A mailbox is a queue of messages with at most one process receiving messages through the mailbox and possibly many senders. The complete operating system specification guarantees that messages sent by a particular process to the same mailbox will be queued in the mailbox in the same order that they were sent but, due to the concurrent nature of the system, does not guarantee the relative ordering of messages sent by different processes. For this reason, Silo specifies a mailbox as a set of queues – one per sender, rather than one per receiver as in miniSilo.

A major challenge in the Silo project is to specify the entire distributed operating system at its interface to user processes, to specify each of the servers and the kernel, and to prove that a composition of the server, kernel and network specifications satisfy the secure distributed operating system specification. We are accomplishing this in stages: first composing the servers that manage mailboxes, then adding the servers that implement system security and support the security features in the programming language.

4 Implementation of MicroSR

4.1 MicroSR Semantics

The interpreted language at this layer is microSR whose constructs include those basic to common sequential programming languages, in addition to an asynchronous *send* statement, a synchronous *receive* statement, a guarded communication *input* statement, and a *co* statement for specifying concurrent execution. This language has the appearance of a high-level system programming language that supports distributed applications. For each statement, we have a semantic transition relation of type $Gstate \rightarrow Gstate \rightarrow Pid \rightarrow Bool$. These semantic relations are analogous to, though more complex than, the mpmachine relations. Here, the type Gstate (for "global state") represents a complete system configuration, and the relation is true if and only if the system may evolve from the first Gstate into the second Gstate by the execution of the given microSR statement within the logical process indicated by Pid. The semantics are also formalized operationally, using multiple copies of a local state abstraction conjoined with a shared pool of messages. These

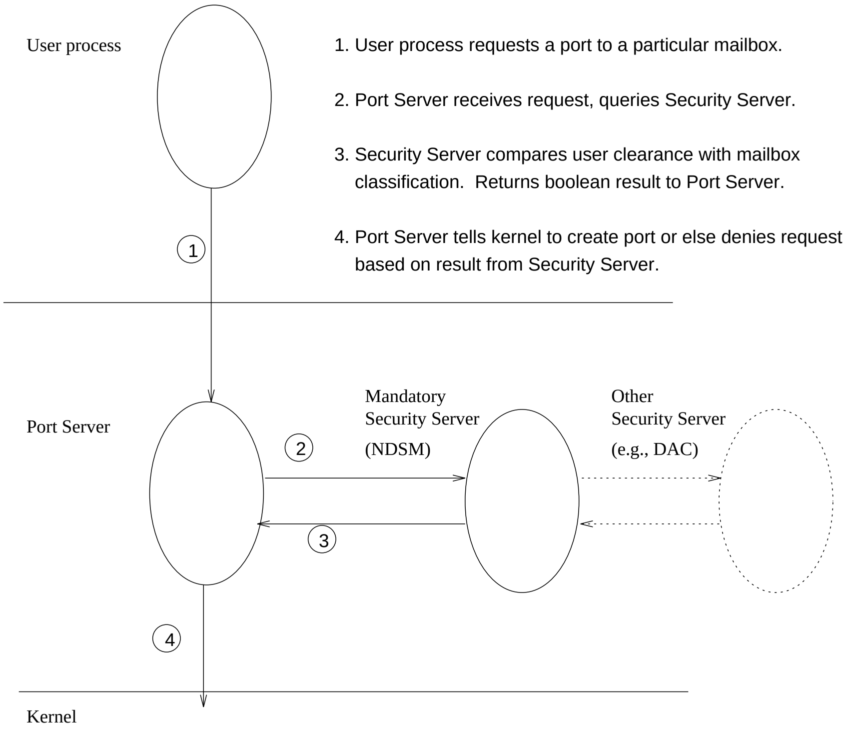

Figure 4: Operating System: Security management

local states are now mappings from variable names into values, rather than register and memory contents. However, the internal structure of this microSR message pool is almost identical to that of the mpmachine — for each program thread, the pool contains a queue of all messages which have been sent to this thread, plus an indication of which ones have been received thus far. To handle security, processes and objects are assigned security levels, and transitions are allowed if they satisfy the standard multilevel security policy.

4.2 Compiler Correctness

Like the previous successful efforts to prove compiler correctness for sequential languages [6, 9], to claim that a compiler is correct is to claim that the target code behavior achieve the source code semantics. Yet, as we have seen, the mpmachine behaviors and the microSR semantics are distinct enough that no canonical equivalence exists between them. We, as the verifiers, must provide this mapping from the abstract microSR global states down into the more concrete mpmachine states. As shown in Figure 5, once this mapping is available, the compiler correctness proof becomes an equivalence proof of two relations, given by the dashed line and the dotted line. For any given starting state, S, of the microSR program, these two relations must agree on which final mpmachine configurations are reachable. In particular, the compiler correctness condition is the following logical equivalence: If, the microSR semantics for the source program indicate that a certain final state, F, is reachable, then it must be true that the mpmachine semantics for the compiler's output code indicate that $F' = \mathit{Mapdown}(F)$ is reachable from $S' = \mathit{Mapdown}(S)$. The compiler itself is simply another mapping function over the domain of legal microSR programs that provides a list of mpmachine instructions for each construct.

A few implementation details are not evident in Figure 5. First of all, since both the Mapdown function and the compiler assign variables to registers, these two assignments must agree in order for

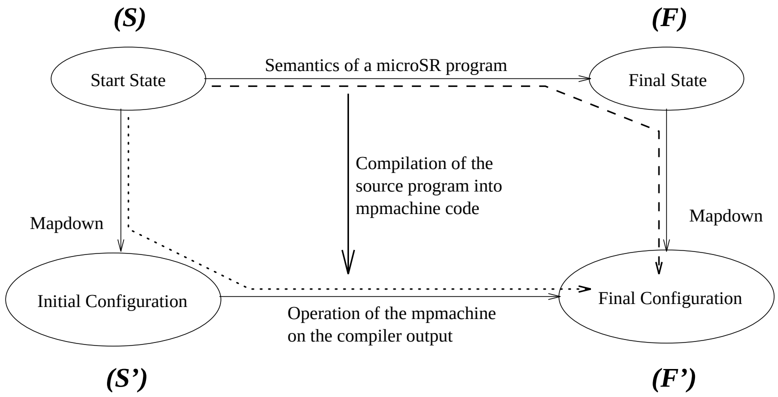

Figure 5: Necessary Mappings for Compiler Correctness

the above equivalence to hold. Consequently, the Mapdown function takes a symbol table argument that indicates the compiler's choices. In miniSilo, there is a fixed symbol table because the microSR language has a small, fixed set of legal variable identifiers. To allow arbitrary strings as identifiers, the compiler needs simply to make an initial pass over the source and gather the necessary symbol table information needed by Mapdown and the second pass. This process involves no concurrency nor composability issues whatsoever other than requiring the extra argument to Mapdown – an aspect that has been accommodated.

As described above, the "dashed" relation, whether true or false, must be equal to the "dotted" relation. This is not entirely possible because a small amount of information is lost across the Mapdown function. For instance, a microSR global state contains a component that indicates the current time of the state. Suppose that we have two states, S_1 and S_2, which are legal starting and ending states for some program. Both the dashed and dotted relations indicate truth. However, suppose that we now alter S_2 ever so slightly, by making its time indicator earlier than that of S_1. Since the global time does not appear in the mpmachine specification, the result of mapping down S_2 is just as it was before, and the dotted relation continues to indicate truth. The dashed relation, however, does not allow for time to decrease, and indicates falsity. The use of the time counter is merely an example; the microSR semantics contain other auxiliary data, such as the number of receives on a particular channel, that are not mapped down to the hardware level. Indeed, when the full Silo contains a kernel with many internal tables, it would not even be clear how the language-level receive counts should be mapped. We do not want the language layer imposing bookkeeping requirements on the kernel, and the correct choice is to not map down the information that is only needed by the language semantics. As a result, the compiler proof is not a complete equivalence, but it must distinguish different means by which the language semantics may indicate falsity.

Similarly, the mpmachine abstraction also contains some items that are not within the image of Mapdown. The first few memory locations are considered to be "reserved" for system use, and the Mapdown function does not dictate the values of the addresses. The fact that the language layer relation holds does not impart any knowledge about this hidden system state within the mpmachine. Consequently, the actual proof requires a third machine configuration (not shown) which is both reachable from S' and equivalent to to F' in all respects except the hidden system state. Finally, within this proof, the complete program is really viewed as a collection of processes, and the picture indicates what must be shown for each individual process. Rather than use fully defined states and configurations, we show that for each process, the relationships of Figure 5 hold amongst that process' view of the system state.

5 MicroSR Applications

5.1 The Hoare Logic for MicroSR

The top application layer is a mechanized Hoare logic for verifying microSR concurrent applications. Our effort to formally derive, using HOL, a sound Hoare logic from microSR semantics is a generalization of similar work by Gordon for a small sequential language [7, 13]. We use semantic relations, rather than functions, in our formal specification for microSR constructs; doing so obviates the possible need for powerdomains in the state abstraction for microSR programs due to the inherent non-determinism. To handle the interference problem arising from concurrent execution, We introduced atomicity and global invariants [2] into our logic system. This logic has been formally proven to be sound within HOL, i.e., axioms and inference rules are all mechanically derived in HOL as the logical implication of the same microSR semantic specification against which the microSR implementation is verified. This logic allows one to reason and state formal assertions about concurrently executing processes that do not share any data objects, but communicate through shared channels that are called operations in SR terminology.

The partial correctness specification in our logic has two levels. The definition of predicate SPEC shown below gives our interpretation of {P_and/or_GI} S {Q_and/or_GI}, the intra-process partial correctness specification, where S is the microSR statement, P and Q are assertions mainly on program variables, GI is the assertion of global invariant mainly on operations, associated with executing S and taken with respect to a particular process. The definition of predicate G_SPEC gives our interpretation of the global partial correctness specification {{(P_list) ∧ GI}} S {{(Q_list) ∧ GI}}, where S is the top level statement for specifying concurrent executions, global invariant GI is the assertion mainly on operations, P_list and Q_list are assertion lists mainly on program variables. The ith elements of the two lists are taken with respect to a particular process for executing the ith sequential program within the top level statement S. Notice that all arguments of SPEC and G_SPEC in the following definitions are abbreviated forms of their meaning functions.

SPEC (P_and/or_GI, S, Q_and/or_GI) = ⊢ *def* ∀ Gstate1 Gstate2 Pid .
    P_and/or_GI(Gstate, Pid) ∧ S(Gstate1, Gstate2, Pid)
    ⇒ Q_and/or_GI(Gstate2, Pid)
G_SPEC ((P_list) ∧ GI, S, (Q_list) ∧ GI) = ⊢ *def* ∀ Gstate1 Gstate2 Pid_list .
    (∀ i . (El i P_list)(Gstate1, (El i Pid_list)) ∧ GI(Gstate, (EL i Pid_list))) ∧
    S(Gstate1, Gstate2, Pid_list)
    ⇒ (∀ i . (El i Q_list)(Gstate2, (El i Pid_list))∧GI(Gstate2, (EL i Pid_list)))

The following gives our representative axioms and inference rules in the derived logic for microSR. Those axioms and rules for microSR sequential constructs, such as the Skip Axioms, Assignment Axiom, If Rule, Do Rule, Sequencing Rule, Precondition Strengthening Rule, and Postcondition Weakening Rule, are not listed below, because their appearance is similar to that in [2, 7], though the way to formally specify and derive them for microSR is actually more complex. All axioms and inference rules are theorems of our language semantics. The "sent-set" σ and "received-set" ρ denote all messages ever sent and received on that channel. $Frontier(\sigma_{\mathrm{op}})$ denotes the earliest message in the channel *op* that has not been received. μ is simply a message constructor function for converting an entity of type integer into one of type message.

- Co Rule

$$\frac{\{GI \wedge Pi\}\ \texttt{SLi}\ \{GI \wedge Qi\}}{\{\{GI \wedge P\_list\}\}\ \texttt{co SL1 // ... // SLn oc}\ \{\{GI \wedge Q\_list\}\}}$$

- Send Axiom

$$\{P \wedge GI \wedge GI^{\sigma_{\mathrm{op}}}_{\sigma_{\mathrm{op}} \cup \mu(\mathrm{E})}\}\ \texttt{send op (E)}\ \{P \wedge GI\}$$

- Receive Rule

$$\frac{P \wedge GI \wedge \mu(\mathrm{E}) \in Frontier(\sigma_{\mathrm{op}}) \Rightarrow Q^{\mathrm{v}}_{\mathrm{E}} \;\wedge\; GI^{\rho_{\mathrm{op}}}_{\rho_{\mathrm{op}} \cup\, \mu(\mathrm{E})}}{\{P \wedge GI\}\ \texttt{receive op(v)}\ \{Q \wedge GI\}}$$

- In Rule

$$\frac{\{P \wedge GI\}\texttt{receive op1(v)}\{R1 \wedge GI\}\texttt{S1}\{Q \wedge GI\}, \{P \wedge GI\}\texttt{receive op2(v)}\{R2 \wedge GI\}\texttt{S2}\{Q \wedge GI\}}{\{P \wedge GI\}\ \texttt{in op1(v) -> S1 [] op2(v) -> S2 ni}\ \{Q \wedge GI\}}$$

5.2 Extensions for Silo

Following our incremental approach, we expect that our final language for Silo will be close to its parent language in its expressive power for distributed computing and our logic will be extended as well. For instance, in our current version of microSR, input statements support only message passing because operations serviced by an input statement can only be invoked by send statements. In our later version, we will allow operations to be invoked by call statements, which will provide rendezvous. We will also extend our input statement with synchronization expressions to allow selective receipt. We will also add some feature into our language to allow users to specify the security level of their programs, resources and processes that they create. The current results at this layer serve as a basis of our research for the complete Silo, since our research so far indicates that SR concurrency features, such as dynamic process creation and that synchronization via message-passing, remote procedure calls, and rendezvous, are all amenable to a Hoare-like programming logic, because the components of our semantic model for microSR have already formalized most of entities and behaviors that SR programmers must consider during their design process. We are now also evaluating the expressive power of our logic by carrying out proofs of programs. The preliminary attempts at manual proof of microSR programs have motivated us to establish a systematic method for creating annotated microSR programs. Another challenging task is to develop, using HOL as well, an interactive prover of LCF [11] style for microSR.

6 Conclusion

Our research on miniSilo has shown how to structure proofs according to vertical layers, how to formally model different layers, how to model the interactions between layers, how to express the proof obligations between layers, and how to compose all the proved layers together. We are extending our research on system design and proof to show that how to evolve miniSilo to Silo in an incremental manner. By our layered proof, we hope to demonstrate that secure and distributed applications can be verified with respect to the entire system, namely showing that microSR applications that are proved correct in our Hoare logic will run correctly on our Silo system.

References

[1] G.R. Andrews, R.A. Olsson, M. Coffin, I.J.P. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, An Overview of the SR Language and Implementation, ACM Transactions on Programming Languages and Systems, 10 (1988) 51-86.

[2] G.R. Andrews, Concurrent Programming: Principles and Practice, The Benjamin/Cummings Publishing Company, Inc. Redwood City, CA, 1991.

[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] W.R. Bevier, and J. Sogaard-Andersen, Mechanically Checked Proofs of Kernel Specifications, in CAV '91, number 575 in Lecture Notes in Computer Science, pp.70-82, Springer Verlag, July 1991.

[5] R. Cardell-Oliver, Using Higher Order Logic for Modelling Real-time Protocols, in TAPSOFT '91, number 494 in Lecture Notes in Computer Science, pp. 259-282, Springer Verlag, April 1991.

[6] P. Curzon, Of What Use is a Verified Compiler Specification, Technical Report No.274, Computer Laboratory, University of Cambridge, November 1992.

[7] M. J. C. Gordon, Mechanizing Programming Logics in Higher Order Logic, in G. Birtwistle and P.A. Subrahmanyam, Eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, New York, 1989.

[8] M. J. C. Gordon and T. F. Melham, Introduction to HOL: A theorem proving environment for higher order logic, Cambridge University Press, Cambridge, 1993.

[9] J.J. Joyce, Totally Verified Systems: Linking verified software to verified hardware. In M. Leeser and G. Brown, Eds., Specification, Verification and synthesis: Mathematical Aspects, Springer-Verlag, 1989.

[10] Z. Manna and A. Pnueli, Verification of Concurrent Programs: A Temporal Proof System, Proc. of the Fourth School of Advanced Programming, Amsterdam, 1982.

[11] L. C. Paulson, Logic and Computation: Interactive Proof with Cambridge LCF, Cambridge University Press, Cambridge, New York, 1987.

[12] V. Yodaiken and K. Ramamritham, Verification of a Reliable Net Protocol, Proc. of the Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerent Systems, number 571 in Lecture Notes in Computer Science, pp. 193-215, Springer Verlag, January 1992.

[13] C. Zhang, R. Shaw, R. Olsson, K. Levitt, M. Archer, M.Heckman, and G. Benson, Mechanizing a Programming Logic for the Concurrent Programming Language microSR in HOL, in J.J. Gordon and C.H. Seger, Eds., Higher Order Logic Theorem Proving and Its Applications, The 6th International Workshop, HUG'93, number 780 in Lecture Notes in Computer Science, pp31-44, Springer-Verlag, March 1994.

[14] MACH 3 Kernel Principles, Open Software Foundation and Carnegie Mellon University, May 1991.

[15] USA INFOSEC Research and Technology, Synergy: A Distributed, Microkernel-based Security Architecture, November 1993.