Understanding and Using Asynchronous Message Passing[*] (Preliminary Version)

Richard D. Schlichting
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, New York 14853

1. Introduction

Message passing provides a way for concurrently executing processes to communicate and synchronize. In this paper, we develop proof rules for asynchronous message-passing primitives (i.e."send no-wait"). Two benefits accrue from this. The obvious one is that partial correctness proofs can be written for concurrent programs that use such primitives. This allows programs to be understood as predicate transformers, instead of by contemplating all possible execution interleavings. The second benefit is that the proof rules and their derivation shed light on how interference arises when message-passing operations are used and on how this interference can be controlled. This provides insight into programming techniques to eliminate interference in programs that involve asynchronous activity. Three safe uses of asynchronous message passing are described here: the transfer of values, the transfer of monotonic predicates, and the use of acknowledgments.

2. Asynchronous Message Passing

The **send** statement

send expr **to** dest

is executed as follows. First, the value of the expression expr[1] is computed. Then, a message with that value is sent tc the process named dest. (Note, we do not prohibit a process from sending a message to itself.)

It is useful to distinguish between _sent_, _delivered_, and _received_ when describing the status of a message. Execution of a **send** statement causes a message to be _sent_. A message that has been sent might subsequently be _delivered_ to its destination. We do not assume that all messages sent are delivered -- real communications hardware does not guarantee reliable delivery of messages. Once delivered, a message can be _received_ by executing a **receive** statement.

A **receive** statement has the form

receive m **when** β

where m is a program variable and β is a Boolean expression involving m and other program variables. Execution of this statement delays the invoker until a message with text MTEXT (say) has been delivered to the invoking process and β^{m}_{MTEXT}= true.[2] Execution of the **receive** completes by assigning MTEXT to m. Thus, β is a guard of the messages that have been delivered; it controls those that can be received.

3. Proof Rules

3.1. Overview

Proofs in our programming logic involve three steps. First, using a sequential programming logic (such as that in [Ho69]) and the communications axioms described below, each process is annotated with assertions, giving a _sequential annotation_. Secondly, assumptions made in the sequential annotation about the effects of

*This work is supported in part by NSF Grant MCS-81-03605.

[1] Both simple and structured values can appear in messages.

[2] The notation β^{m}_{MTEXT} denotes the substitution of MTEXT for every free occurrence of m in β.

receiving messages are validated by showing satisfaction [Le81]. This involves constructing a collection of satisfaction formulas and proving them to be valid. Finally, non-interference [Ow76] is established to ensure that execution of no process can invalidate assertions that appear in the sequential annotation of another. In the following, each step is treated in detail.

3.2. Communications Axioms

When buffered, asynchronous communication primitives are used, the state of the system includes information about messages that are in transit. In order to make assertions about that aspect of the system state, two multisets[3] are associated with each process D. The send multiset for process D -- denoted σ_D -- contains a copy of every message that has been sent to D. Similarly, the receive multiset for process D -- denoted ρ_D -- contains a copy of every message that has been received by D. A message can be received only if it has been sent and delivered. Therefore:

Network Axiom: $(\forall D: D \text{ a process}: \rho_D \subseteq \sigma_D)$

The effect of executing

send expr **to** D

is the same as that of executing the assignment

$$\sigma_D := \sigma_D \oplus \text{expr}$$

where "$\sigma_D \oplus$expr" denotes the multiset consisting of the elements of σ_D plus an element with value expr. Using the weakest precondition predicate transformer (wp) [Di76] with respect to the postcondition $T \wedge \sigma_D = \sigma_0 \oplus \text{expr}$, we get an axiom for the **send** statement:

Send Axiom:

$\{T^{\sigma_D}_{\sigma_D \oplus \text{expr}} \wedge \sigma_D = \sigma_0\}$ **send** expr **to** D $\{T \wedge \sigma_D = \sigma_0 \oplus \text{expr}\}$

When execution of the statement

receive m **when** β

in process D terminates, β is true and a copy of the message received is in ρ_D. In addition, depending on the particular message received, it may be possible to make some assertion about the state of the sender. An axiom that captures this is:

Receive Axiom:

$\{R \wedge \rho_D = \rho_0\}$ **receive** m **when** β $\{\rho_D = \rho_0 \oplus m \wedge \beta \wedge Q\}$

At first it may be disturbing that following a **receive** statement anything can be asserted, as indicated by the miraculous appearance[4] of Q in the postcondition. In the course of establishing satisfaction, restrictions are imposed on Q.

3.3. Establishing Satisfaction

Given a distributed program made up of processes $D_1, D_2, \ldots, D_n$, assume each process has been annotated using a sequential programming logic and the communications axioms above. In a sequential annotation, the assertion immediately preceding a statement s will be referred to as the precondition of s and will be denoted pre(s); the assertion following s will be referred to as the postcondition of s and will be denoted post(s). Assertions involve auxiliary variables [Cl73], program variables and the multisets σ_D and ρ_D for each process D. Distinct variables are assumed to have distinct names. The values of program variables are stored in memory; auxiliary variables need not be, since, by definition, their values do not influence execution of processes. Accordingly, program variables can be named in assertions in the sequential annotation of any process, but can appear only in statements in processes that have access to the memory in which those variables are stored. Allowing assertions in one process to refer to variables that are accessible only in another allows the states of different processes to be correlated.

Consider a **receive** statement

r: **receive** m **when** β

in process D. In order for execution of r to result in the receipt of a message with text MTEXT, then (1) β^m_{MTEXT} must be true and (2) a message with text MTEXT must have been sent to D but not yet received. Thus, immediately before MTEXT is assigned to m, the system state can be characterized by

$$\text{pre}(r) \wedge \beta^m_{\text{MTEXT}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)$$

[3]A multiset -- sometimes called a "bag" -- is like a set, but can contain more than one instance of the same element.

[4]See [Di76] for a discussion of the "Law of the Excluded Miracle".

where $\ominus$ is the multiset difference operator.

Execution of r resulting in receipt of a message with text MTEXT is equivalent to execution of the following multiple assignment statement:

$$m, \rho_D := \text{MTEXT}, \rho_D \oplus \text{MTEXT}.$$

For this assignment to establish the postcondition of the Receive Axiom, execution must be performed in a state satisfying

$$wp(\text{"}m,\rho_D := \text{MTEXT}, \rho_D \oplus \text{MTEXT"}, \rho_D = \rho_0 \oplus m \wedge \beta \wedge Q)$$

which is

$$= (\rho_D = \rho_0 \oplus m \wedge \beta \wedge Q)^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$$

$$= \rho_D = \rho_0 \wedge \beta^m_{\text{MTEXT}} \wedge Q^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}.$$

Thus, the Receive Axiom will be sound with respect to our operational model provided

$$(\text{pre}(r) \wedge \beta^m_{\text{MTEXT}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow (\rho_D = \rho_0 \wedge \beta^m_{\text{MTEXT}} \wedge Q^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}})$$

is true. Because $\text{pre}(r) \Rightarrow \rho_D = \rho_0$, this is equivalent to

Sat(MTEXT):

$$(\text{pre}(r) \wedge \beta^m_{\text{MTEXT}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow Q^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$$

If Sat(MTEXT) is true whenever r is executed, then receipt of a message with value MTEXT will leave post(r) true. Possible values for MTEXT can be characterized as follows. Consider each **send** statement s that names D as its destination:

s: **send** expr **to** D.

The message sent by executing s is determined by evaluating expr in the state that exists when s is executed. Let $\overline{id}$ represent the list of program and auxiliary variables, and $\overline{v}$ be a vector of values corresponding to an assignment of values to the elements of $\overline{id}$. $\overline{v}$ is used to model the state of the system when s is executed. Possible values for MTEXT resulting from execution of s are defined by

$$\text{pre}(s)^{\overline{id}}_{\overline{v}} \wedge \text{MTEXT} = \text{expr}^{\overline{id}}_{\overline{v}}$$

Thus, we can define the Asynchronous Satisfaction Formula for s and r.

$\text{Satisfaction}_{\text{asynch}}(s,r)$:

$$(\text{pre}(s)^{\overline{id}}_{\overline{v}} \wedge \text{MTEXT} = \text{expr}^{\overline{id}}_{\overline{v}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D) \wedge \text{pre}(r) \wedge \beta^m_{\text{MTEXT}}) \Rightarrow Q^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$$

Provided $\text{Satisfaction}_{\text{asynch}}(s,r)$ is true when r is executed, receipt of a message sent by executing s will leave post(r) true. A **send** statement that names process D as its destination will be said to match every **receive** statement in D. Thus, for every **send** s that matches r, the truth of $\text{Satisfaction}_{\text{asynch}}(s,r)$ when r is executed ensures that post(r) will be true, if r terminates.

Unfortunately, pre(r) may not contain enough information to prove that $\text{Satisfaction}_{\text{asynch}}(s,r)$ is true in that state, because Q can involve variables manipulated by the sender. Assertions in the sequential annotation of the sender may contain the required information. Since the exact state of the sender is in general unknown when a message is received, we must show that $\text{Satisfaction}_{\text{asynch}}(s,r)$ is true in any state of the sender. This yields:

Asynchronous Satisfaction Rule
For every **send** statement s and **receive** statement r that match, and every assertion A in the sequential annotation of the process containing s, the following must be proved valid:

$$A \Rightarrow \text{Satisfaction}_{\text{asynch}}(s,r)$$

3.4. Establishing Non-Interference

Since assertions in one process can refer to variables changed by another, it is necessary to show that execution of no process invalidates assertions in the proof of another. This is called non-interference [Ow76].

An assertion P and statement s' are *parallel* if s' is contained in one process and P is contained in the proof of a different process. To establish non-interference, it must be shown that execution of every assignment, **receive** and **send** statement parallel to P leaves P unchanged. We shall say that *s' does not interfere with P* if:

(3.4.1) $\{P \wedge \text{pre}(s')\}\ s'\ \{P\}$

is a theorem. Non-interference is established by proving that no statement s' interferes with any assertion that is parallel to it.

For s' a **receive** statement r, (3.4.1) is trivial because of the Receive Axiom. Satisfaction must be established for this additional theorem. For each matching **send** statement s of the form:

s: **send** expr **to** D

a satisfaction formula is constructed

(3.4.2)

$$(pre(s)_{\overline{v}}^{\overline{id}} \wedge MTEXT=expr_{\overline{v}}^{\overline{id}} \wedge MTEXT \in (\sigma_D \ominus \rho_D) \wedge (P \wedge pre(r)) \wedge \beta_{MTEXT}^{m}) \Rightarrow P_{MTEXT,\rho_D \oplus MTEXT}^{m,\rho_D}.$$

Then, for each assertion A in the sequential annotation of the process containing s, A $\Rightarrow$ (3.4.2) must be proved valid.

This additional satisfaction proof is necessary because a **receive** is, in effect, a (decentralized) assignment statement. Proof that this assignment does not interfere with assertions that are parallel to it is therefore necessary.

4. Safe Uses of Asynchronous **send**

Establishing satisfaction is not always a simple task. In this section, we explore disciplined uses of asynchronous message-passing primitives for which satisfaction is easily established.

4.1. Restricted Postconditions

Execution of a process might depend on the value of a message received, and not on the state of the sending process at the time that message is received. In such cases, the postcondition of the **receive** statement will not refer to variables in the sender's state, but will instead be solely in terms of the message value. Because of this, execution by the sender cannot invalidate the postcondition of the **receive**, and the corresponding satisfaction formula will be a tautology, hence valid.

4.2. Monotonic Preconditions

For a variety of reasons, it may be necessary following receipt of a message for execution of the receiving process to be synchronized with that of the sender. Thus, an assertion about the state of the sender would appear in the postcondition of the **receive**. In addition to transferring values from sender to receiver, such a **receive** facilitates transfer of a predicate, called the transferred predicate, from the proof of the sender to the proof of the receiver. Transfer of a predicate must be done with care so that subsequent execution by the sender does not invalidate the postcondition of the **receive** before the message is received.

An assertion is monotonic if once it becomes true it remains so. The use of monotonic preconditions for **send** statements facilitates establishing satisfaction. If the sending process is structured so that the transferred predicate is implied by the precondition of the **send** statement and by every subsequent state, it will be true when the message is received, regardless of delivery delays. Hence, the predicate can appear in the postcondition of **receive** statement. In general:

Monotonic Preconditions for Send Statements
> Establishing satisfaction between **send** s and **receive** r follows from the invariance of
>
> I_{sender}: "message not sent" $\vee$
> "transferred predicate true"

when pre(s) implies the portion of post(r) that is a transferred predicate and the transferred predicate is monotonic in the sender.

4.3. Acknowledgment Messages

It is also possible to transfer non-monotonic predicates between processes. Then, the structure of the sending process must ensure the truth of the transferred predicate when the message is received. Since there is no way of knowing exactly when that time is, the receiving process must transmit an acknowledgment message to communicate that fact. Hence, sending acknowledgments can be viewed as a way to ensure the validity of a satisfaction formula: between the time the message is sent and its acknowledgment is received, the sender keeps the transferred predicate true, so that between the time the message is received and the acknowledgment sent the receiver can assert the transferred predicate.

The general rule, then, is:

Non-monotonic Predicates
> A message can be used to transfer a predicate from process S to process R as long as S ensures that the predicate is true at the time the message is received. This can be done by ensuring that the predicate is true at the time the message is sent and remains true un-

til R informs S that the predicate is no longer required by acknowledging it. Thus, the sender maintains the following:

I_{sender}: "message not sent" $\vee$
"acknowledgment received" $\vee$
"transferred predicate true".

4.4. Satisfaction Revisited

Establishing satisfaction ensures that the postcondition of a **receive** will indeed be true if a message is received. Any use of asynchronous message-passing operations that keeps the corresponding satisfaction formula true at all times is safe. Each of the three techniques described above did just that, by ensuring that the sender maintained an invariant relation which happened to imply the validity of the corresponding satisfaction formula. For the satisfaction formula corresponding to a **send** s and a **receive** r, these invariants were such that each assertion in the sender implies:

(1) The message that could be received by r from s had not yet been sent or had already been received. Thus, $MTEXT \in (\sigma_D \ominus \rho_D)$, a conjunct in the antecedent of the satisfaction formula, is false.

(2) A message has been sent but cannot be received. Thus, pre(r) or β^m_{MTEXT} in the antecedent of the satisfaction formula is false.

(3) A message could be received and $Q^{m,\rho_D}_{MTEXT,\rho_D \oplus MTEXT}$ is true. Thus, the consequent of the satisfaction formula is true.

Establishing satisfaction is equivalent to proving that the sender will not invalidate the postcondition of a **receive**. One might expect this obligation to be superfluous, arguing that interference with the postcondition of a **receive** statement by a sender should be detected when performing a non-interference proof. Unfortunately, because messages are buffered, a statement s' in one process can interfere with the postcondition of a **receive** even if s' cannot be executed while the receiver is waiting for a message. To see this, consider the following sequential annotation.

```
SELF: process
        x:= 2;
        {x=2}
        send "x_is_two" to SELF;
        x:= 3;
        {x=3}
        receive m when true;
        {m="x_is_two" ∧ x=2}
        end
```

Here, a process SELF sends a message to itself and then invalidates the transferred predicate (x=2) before executing a **receive**. The sequential annotation is correct, since it can be derived from our axioms. There is only one process, so interference freedom is trivially established. Yet, the postcondition of the **receive** will not be true when the **receive** terminates. An attempt to establish satisfaction, however, will fail. (It is possible to construct similar pathologies for programs involving more than one process as well.)

In fact, establishing satisfaction is a form of non-interference proof. Establishing satisfaction in concert with performing a non-interference proof is equivalent to showing that no statement s' interferes with the satisfaction formula, a global invariant.

5. Discussion

5.1. Related Work

Axioms for reasoning about buffered asynchronous message-passing were first proposed in connection with Gypsy [Go79]. There, **send** and **receive** are characterized in terms of their effects on shared, auxiliary objects called buffer histories. The proof rules for **send** and **receive** are derived from the assignment axiom by translating these statements into semantically equivalent assignments to buffer histories. Because in Gypsy program variables manipulated by one process may not appear in assertions in the proof of another, there is no need to perform satisfaction or non-interference proofs. Unfortunately, this also restricts the class of programs that can be proved correct to those in which (only) values are transferred; programs in which predicates are transferred by message-passing cannot be proved. Nevertheless, Gypsy has been successfully used to verify several large concurrent systems.

Our work extends the notion of satisfaction [Le81] to asynchronous message passing. In [Ap80], a proof system for synchronous message passing is presented that requires the construction of a global invariant. While this is like our notion of a satisfaction formula, the construction of a satisfaction formula is a mechanical task, hence simpler than the cooperation proof required in [Ap80]. We also feel that our satisfaction formulas give the programmer insight into how to effectively use message-passing statements. A different approach to proving correctness of programs that use message passing is based on the use of traces of communications actions. This approach is explored in [Mi81] for synchronous message passing; extension to asynchronous message passing remains an open problem.

5.2. Variations on a Theme

The message-passing primitives we have considered can be used to model any of a variety of existing mechanisms. While it is unlikely that anyone would actually implement **receive** as we have defined it -- the expense and complexity of "peeking" at the contents of delivered messages in order to decide if the Boolean expression would be satisfied is too high -- by suitable choice of the Boolean β, our **receive** can be used to model communications primitives in which messages are received in the order sent. Using **receive**, it is also possible to model receive statements that have time-outs [Sc82a] and to implement higher-level communications primitives such as remote procedure calls and synchronous message passing as is described in [Sc82b].

The syntax of **send** requires that a destination process be explicitly named. This was done so that construction of the satisfaction formulas could be a mechanical task. However, our approach can be generalized to handle the case where a communications channel is named as a destination. Then, satisfaction formulas would be constructed for every **send-receive** pair that reference the same channel. The fact that fewer satisfaction formulas should result (due to the reduced number of pairs), is a strong formal argument in favor of including such a channel facility in a programming notation. It is also possible to handle the case where the destination in a **send** is computed at runtime: a satisfaction formula for each possible destination process would then have to be constructed and proved valid.

Lastly, in some programming notations **receive** statements can appear in the guards of selection and iteration statements. A guard containing a **receive** statement r is true only if execution of r would not be delayed. This allows a program to wait for the occurrence of any one of a number of asynchronous events. From the viewpoint of partial correctness, the guarded command

$$r \rightarrow S$$

where S is a statement and r a **receive** is equivalent to

$$\mathrm{true} \rightarrow r;S.$$

The only difference is that the latter is more prone to deadlock than the former[5]. Since termination and deadlock are not reflected in our proof system, it is possible to handle such constructs by translating them into "equivalent" statements that do not include input commands in guards.

5.3. Miracles and Divine Inspiration

The satisfaction requirements are indicative of an additional difficulty of understanding programs that communicate and synchronize using message passing: processes cannot be viewed in isolation. The sequential axiom for **receive** allows anything to be concluded following receipt of a message by process D. However, choosing postconditions for which satisfaction can be established requires knowledge about the actions of all processes that send messages to D -- not only knowledge about the values of the messages that can be received, but also knowledge about the states of the senders.

In light of this, the fact that each process can reference only its local variables becomes an impediment to designing a distributed program. Primitives intended for communication and synchronization with shared memory allow processes to interrogate each others' states to effect synchronization. Usually, the condition being awaited appears in the text of the synchronization mechanism. For example,

await semph>0 **then skip**

delays its invoker until variable semph has a non-negative value. The appearance of the condition being awaited -- semph>0 -- in the text of the statement makes it easier to construct a process in isolation; assertions in the proof can be deduced from the program text. Thus, the system need be considered as a whole only when establishing non-interference. The same is not true of a distributed system. Although the exchange of messages allows processes to synchronize, a

[5]This was first pointed out in [Le81].

condition being awaited may not appear in the text of a **receive** statement. Recall that the Boolean expression in a **receive** can refer only to local variables and to messages that have been delivered but not yet received. If the condition being awaited is a transferred predicate and there is no shared memory, then it is impossible for the receiver to directly interrogate the awaited condition. Thus, the awaited condition cannot appear in the text of the **receive** statement, and so the programmer must inspect all potential senders to make sure that termination of the **receive** occurs only if the awaited condition is true.

In summary, the absence of shared memory complicates the programming task without a corresponding simplification in the proof. The programmer of each process has access to only a small part of the system state, but the proof might well include assertions about the entire system state. This requires that the programmer understand the global state of the system.

6. Conclusion

Proof rules for asynchronous message passing have been presented. The proof rules allow programs written using **send** and **receive** to be formally verified. They also give insight into safe ways to use asynchronous message-passing statements.

From a methodological view point, distributed programs are not fundamentally different from concurrent programs that use shared memory. Both require interference to be identified and controlled. Additional complexity in the case of a distributed program stems from two factors. First, in a distributed program the programmer of a process has access to only a small part of the system state, yet must reason about the entire system state. Secondly, the programmer must maintain the truth of additional global invariants -- the satisfaction formulas.

The programming logic described in this paper allows proofs of partial correctness. In addition, one might want to prove that a program is free from deadlock or has certain liveness properties. Extensions to facilitate proving such properties are under currently under investigation.

Acknowledgments

Discussions with Alan Demers, David Gries, Leslie Lamport and Gary Levin have been invaluable in formulating the ideas in this paper. We are also indebted to Bowen Alpern, Greg Andrews and Ozalp Babaoglu for helpful comments on an earlier draft of this paper.

References

[Ap80] Apt, K., N. Francez, and W. DeRoever. A Proof System for Communicating Sequential Processes. TOPLAS 2, 3 (July 1980), 359-385.

[Cl73] Clint, M. Program Proving: Coroutines. Acta Informatica 2, 1 (1973), 50-63.

[Di76] Dijkstra, E.W. A Discipline of Programming. Prentice Hall, 1976.

[Go79] Good, D., R. Cohen, and J. Keeton-Williams. Principles of Proving Concurrent Programs in Gypsy. In Proceedings of the Sixth Annual Symposium on Principle of Programming Languages (January 1979), 42-52.

[Ho69] Hoare, C.A.R. An Axiomatic Basis for Computer Programming. CACM 12, 10 (October 1969), 576-580.

[Le81] Levin, G., and D. Gries. Proof Techniques for Communicating Sequential Processes. Acta Informatica 15 (1981), 281-302.

[Mi81] Misra, J., and K. Chandy. Proofs of Networks of Processes. IEEE Trans. on Software Eng. SE-7, 4 (July 1981), 417-426.

[Ow76] Owicki, S., and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. Acta Informatica 6 (1976), 319-340.

[Sc82a] Schlichting, R.D. Axiomatic Verification to Enhance Software Reliability. Ph.D. Thesis, Cornell University, January 1982.

[Sc82b] Schlichting, R.D. and F.B. Schneider. Using Message Passing for Distributed Programming: Proof Rules and Disciplines. Submitted to TOPLAS.