Working Paper 187

May 1979

Towards a Better Definition of Transactions

Barbara S. Kerns

This report describes research done at the Artificial Intelligenc Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Office of Naval Research of the Department of Defense under Contract N00014-75-C-0522.

Artificial Intelligence Laboratory

Massachusetts Institute of Technology

Towards a Better Definition of Transactions

by

Barbara S. Kerns

Abstract

This paper builds on a technical report written by Carl Hewitt and Henry Baker called "Actors and Continuous Functionals". What is called a "goal-oriented activity" in that paper will be referred to in this paper as a "transaction". The word "transaction" brings to mind an object closer in function to what we wish to present than does the word "activity".

This memo, therefore, presents the definitions of a reply and a transaction as given in Hewitt and Baker's paper and points out some discrepancies in their definitions. That is, that the properties of transactions and replies as they were defined did not correspond with our intuitions, and thus the definitions should be changed. The issues of what should constitute a transaction are discussed, and a new definition is presented which eliminates the discrepancies caused by the original definitions. Some properties of the newly defined transactions are discussed, and it is shown that the results of Hewitt and Baker's paper still hold given the new definitions.

I. Introduction

A transaction corresponds to our usual notion of a subcomputation needed for subroutines. It includes those events which occur because a certain request is made, up to and including the resultant reply. The notion of a request, followed by steps leading to a reply, appears over and over again in many different kinds of programming applications. Recursive function invocation, data bases, and interactive systems, for example, each illustrate the need for the concept of a transaction. In recursive function invocation a request is made for the value of some expression, and a reply is subsequently returned. When working with data bases, one often wishes to retrieve a piece of information and thus will submit a request. Here again, the activity involved in replying to that request constitutes a transaction. Interactive systems are really nothing more than a series of requests and replies. Lisp, for example, uses the classic "read-eval-print" loop.

The concept of a transaction is therefore an important one, and is extremely useful in reasoning about sequential program semantics. We need to establish a robust definition of a transaction that applies to distributed systems as well, where many machines or processors interact through a network. Communication between processes is necessary for concurrent programming to be useful; thus we wish to construct and examine a definition of a transaction which can be used to reason about such inter-process communication.

II. Background

Actors and events are the basic concepts of the actor theory. Actors communicate with one another by sending messengers to each other. Each messenger contains information which the receiving or "target" actor then acts upon. An actor may create another actor, in fact, most messengers (which are also actors) are created just before being sent off to another actor. An event occurs when a messenger arrives at its target actor. Often we use the notation:

$$E: [T <\sim\sim M]$$

to mean that target(E) = T and messenger(E) = M.

Actors which a given actor directly knows about are called its "acquaintances". For an event E, the "participants" of E are the target(E), the messenger(E), and the acquaintances of target(E) and of messenger(E). An actor maintains a vector of acquaintances, which may or may not change over time. It may gain new acquaintances (or forget old ones) through the acquaintances of a message sent to it. An example of an actor whose acquaintances change over time is a "cell". It has one acquaintance, and can receive either a "contents?" request, in which case it replies with its acquaintance, or a update request, in which case it forgets its old acquaintance and remembers the new one given to it by the update request. The behavior of other actors whose vectors of acquaintances may change with time are given in [Hewitt and Attardi, 1978].

The significance of an event causing an actor to change its vector of acquaintances is that such actors therefore are "order-dependent". That is, the order in which they receive messages can effect the replies they send to these messages. Such actors are "serialized" so that they can assign an "arrival ordering" to their messengers. If the message of event E_1 arrives at a serialized actor S before the message of event E_2, then we write:

$$E_1 \text{ -}arr_S\text{ -> } E_2$$

Another type of ordering is the "activation ordering". If as a result of receiving a messenger M in an event E_2 the target actor sends another messenger M_2 to an actor A, then E_2 is said to activate E_3 where E_3 is the arrival of M_2 at A. We write:

$$E_2 \text{ -}act\text{-> } E_3$$

The transitive closure of these two kinds of orderings is called the "combined ordering", and according to the above two examples we could write:

$$E_1 \text{ ---> } E_3$$

II. Transactions

II. i. Request and Reply Events

In order to study transactions we must have a formal definition of a request and a reply. A request is simply the messenger in any event of the form:

[... <~~ [request: ...,reply-to: c]]

where c is a continuation. The definition of reply as given in [Hewitt and Baker, 1977] is:

> If an event E is of the form
> [... <~~ [request: ...,reply-to: c]]
> then any event E' of the form
> [c <~~ [reply: ...]]
> such that E-*act*->E' will be said to be a reply to E.

(We will frequently refer to an event whose messenger is a request or a reply as a request or reply event, respectively. We use the notation "reply(RQ)" to mean the event whose messenger is the reply of the request event RQ. This paper assumes that at most one reply exists for each request.) But this definition of a reply is too strict. Consider the case in which a request is sent to a serialized actor X in event RQ. Suppose that before sending a reply, X demands that it receive "permission" to do so. Permission is granted in the form of the receipt of a clock pulse, which may arrive before or after the receipt of the request event. Calling the event in which the clock pulse arrives at X event E, we have E: [X <~~ pulse]. The pulse allows the reply to the first message to be sent, and it arrives at the continuation in event RP, such that RP: [C <~~ [reply: ...]].

RQ: [X <~~ [request: M, reply-to: C]]
arr_X
E: [X <~~ pulse] -*act*-> RP: [C <~~ [reply: ...]]

We see that RQ-*arr*$_X$->E-*act*->RP. There is no activation ordering between RQ and RP, but RP should still constitute a reply to RQ. We therefore propose that the definition of reply be weakened to:

> If an event E is of the form
> [... <~~ [request: ...,reply-to: c]]
> then any event E' of the form
> [c <~~ [reply: ...]]
> such that E--->E' will be said to be a reply to E.

By changing the requirement of an activation ordering between the request and its associated reply to a combined ordering, we allow events which are ordered by arrival ordering to enter the path between request and reply.

II. ii. Redefining Transactions

Hewitt and Baker's definition of a transaction (given this paper's assumption that at most one reply exists for each request) is:

$$\text{transaction}(RQ) = RQ \dashrightarrow \cap \dashrightarrow \text{reply}(RQ)$$

where RQ is an event whose messenger is a request.

Intuitively, a transaction is an attempt to characterize the notion of a process in conventional programming languages, since only those events which contribute towards the request's reply are included in the transaction.

For example, consider an event RQ_1 in which a message M arrives at a serialized actor X with continuation C, that is, RQ_1: [X <~~ [request: M, reply-to: C]]. Let X then receive a second message M', such that RQ_2: [X <~~ [request: M', reply-to: C']]. X replies to M' first by sending R' to the continuation C'. It then replies to M. The following events and orderings are relevant.

RQ_1: [X <~~ [request: M, reply-to: C]]
RQ_2: [X <~~ [request: M', reply-to: C']]
RQ_1 -arr_X-> RQ_2
RP_2: [C' <~~ R']
RP_1: [C <~~ R]
RQ_1 -*act*-> RP_1
RQ_2 -*act*-> RP_2

RQ_1: [X <~~ [request: M, reply-to: C]] -*act*-> RP_1: [C <~~ R]
|
arr_X
↓
RQ_2: [X <~~ [request: M', reply-to: C'] -*act*-> RP_2: [C' <~~ R']

Now, transaction(RQ_1) = {RQ_1, RP_1}, and transaction(RQ_2) = {RQ_2, RP_2}. Although RQ_1--->RQ_2, RQ_2 is not an element of transaction(RQ_1), because it is not true that RQ_2--->RP_1.

However, as originally pointed out by Craig Schaffert, we note that a discrepancy can arise with this definition. Consider the case in II.i. above in which a clock pulse was used to activate the reply to a request. According to Hewitt and Baker's definition of a transaction, transaction(RQ) = {RQ, E, RP}. But if the clock pulse arrives at X before RQ, we have the following situation:

E -arr_X-> RQ -*act*-> RP

Now transaction(RQ) = {RQ, RP}. This raises several questions concerning just what should be included in a transaction. Should E be included in the transaction in either case? Should it not? Should the whole computation fail to be recognized as a transaction?

In keeping with our intuitive discussion of transactions, it seems that we shouldn't throw out the whole computation, but we must now decide whether E should be included or not, and in either case, its inclusion or exclusion should be consistent and not dependent on the arrival ordering of E.

Carl Hewitt has proposed that those events which are not request events or reply events (where a reply is extended to include complaints), should not be allowed to be members of any transaction. This constraint is in keeping with our concept of a transaction as that "thing" which models the classical notion of a process as a set of nested request events and events which reply to those requests.

Adding this constraint to our definition of a transaction, we see that in order to determine whether E should be included in transaction(RQ), we must know whether it constitutes a request event or not (clearly it is not replying to X). If E is not a request event, then E will not be a member of transaction(RQ) regardless of where it comes in the arrival ordering of X with respect to RQ. However, if E is a request event, it necessarily has an associated reply event. We will assume then that there is an event R such that R = reply(E). We can now put some constraint on R in order to include or exclude E (and R) from transaction(RQ). Following our intuitions (this is a definition, after all), we add the constraint that if E is a request, in order for E to be an element of transaction(RQ), R--->reply(RQ). More formally, we now have:

> For some request or reply event E', E' ∈ transaction(RQ) iff
> RQ--->E', E'--->reply(RQ), and if E' is a request event,
> then reply(E')--->reply(RQ).

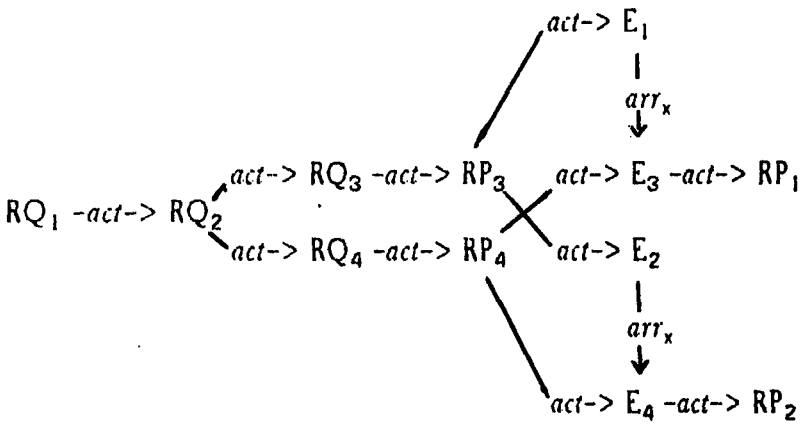
What this means is that if a request event is to be part of a transaction, its associated reply event should be also. For the clock pulse example then, transaction(RQ) = {RQ, RP} where E is not included at all, since E is neither a request nor a reply event. Note that since we have added constraints to the definition of transaction but not eliminated any, that no event which was not part of a given transaction will now be defined to be. We have only eliminated certain "ad hoc" events from some transactions. We will examine later how this effects some of the results presented in [Hewitt and Baker, 1975].

II. iii. Properly Nested Transactions

We would now like to prove some properties of these transactions. In particular, it would be nice to be able to say that transactions are "properly nested". That is, that two transactions are either disjoint, or that one is a subset of the other. Unfortunately, a counter-example follows.

Consider the following event network and its associated orderings (Where the RQ's are request events, the RP's are reply events, and the E's are neither. RP's correspond to the RQ with the same subscript):

RQ_1 -*act*-> RQ_2
RQ_2 -*act*-> RQ_3, RQ_4
RQ_3 -*act*-> RP_3
RQ_4 -*act*-> RP_4
RP_3 -*act*-> E_1, E_2
RP_4 -*act*-> E_3, E_4
E_1 -arr_X -> E_3
E_2 -arr_X -> E_4
E_3 -*act*-> RP_1
E_4 -*act*-> RP_2



We wish to determine which events are members of transaction(RQ_1) and which are members of transaction(RQ_2). Transaction(RQ_1) consists of RQ_1 (obviously), but not RQ_2, since RP_2 = reply(RQ_2) has no ordering with respect to RP_1 = reply(RQ_1). RQ_3 and RQ_4 are both elements, since they are ordered with respect to RQ_1 and RP_1, and their respective replies precede RP_1. Then their replies RP_3 and RP_4 are also members. E_1

through E_4 are not members of transaction(RQ_1) since they are neither request events nor reply events. Finally, RP_1 is a member of transaction(RQ_1). Therefore, transaction(RQ_1) = {RQ_1, RQ_3, RQ_4, RP_3, RP_4, RP_1}. Similarly, transaction(RQ_2) = {RQ_2, RQ_3, RQ_4, RP_3, RP_4, RP_2}.

The intersection of transaction(RQ_1) with transaction(RQ_2) consists of four events, {RQ_3, RQ_4, RP_3, RP_4}, and although this set is not a transaction itself, it consists of the union of two transactions. (It is possible to show that the intersection of two transactions is always equal to the union of some number of other transactions.) However, transaction(RQ_1) is clearly not contained in transaction(RQ_2), nor is transaction(RQ_2) contained in transaction(RQ_1).

Well, all is not lost, for we *can* prove at least a slightly weaker property, though one which is still quite useful. Though we can not show that given any two transactions with at least one event in common, one transaction must be contained in the other, we can show that if a request event RQ is an element of transaction(RQ'), then transaction(RQ) $\subseteq$ transaction(RQ'). This is called the *Law of Containment for Transactions.*

Assume that E $\in$ transaction(RQ). To show that E $\in$ transaction(RQ'), we must show

Goal 1: RQ'--->E--->reply(RQ')

and if E is a request event, that

Goal 2: reply(E)--->reply(RQ').

Since E $\in$ transaction(RQ) we have:

RQ--->E--->reply(RQ)

and since RQ $\in$ transaction(RQ'):

RQ'--->RQ--->reply(RQ)--->reply(RQ').

Then

RQ'--->RQ--->E--->reply(RQ)--->reply(RQ')

Thus RQ'--->E--->reply(RQ'), which proves Goal 1.
Assume E is a request event in order to prove Goal 2:

reply(E)--->reply(RQ').

Since E ∈ transaction(RQ) then

reply(E)--->reply(RQ),

and since RQ ∈ transaction(RQ'), and RQ is a request event,
we know

reply(RQ)--->reply(RQ').

Thus reply(E)--->reply(RQ'). [Done]

III. Continuous Functionals

III. i. Continuation Ordering

Before we go on, let's briefly characterize those events which we have eliminated from transactions. First of all, we have eliminated from transactions all those events which are neither request nor reply events. Secondly, we have eliminated all those request events whose associated reply events do not also participate in the transaction.

Hewitt and Baker have defined a third ordering on events called the continuation ordering. In this ordering, E_1 *-cont->* E_2 if 1) there is some transaction α such that E_1 and E_2 are both members of α, and 2) E_1 ---> E_2. Our redefinition of transaction affects this ordering to the extent that now if E_1 *-cont->* E_2, we may automatically conclude that E_1 and E_2 are either request or reply events since no other

type of event may be an element of some transaction, and furthermore, given the ordering RQ_1 *-cont->* RQ_2, we can conclude reply(RQ_2) *-cont->* reply(RQ_1). It is also the case that some continuation orderings that once held between two events may no longer hold, since some events have been eliminated from transactions. But no additional continuation orderings will hold due to the redefinition of transaction.

III. ii. Fork and Join Behavior

The fork and join behavior discussed in Section IX of [Hewitt and Baker, 1975] holds up beautifully under the new definition of transaction, as long as no join occurs without a previous fork first providing the components of the join. This prerequisite is easy to fulfill, however, since the classic notion of a process implies that that is always the case.

III. iii. Procedures and Mathematical Functions

The definition of a procedure as given in [Hewitt and Baker, 1975] requires that 1) all events involved in the procedure are either request or reply events, 2) there is at most one reply event for each request event, and 3) the transactions are properly nested. That is, for any two transactions in the procedure, either one is a proper subset of the other, or they are disjoint.

We wish to show that any transaction which was a procedure under the old definition is still a procedure under the new definition. That is, we wish to show that any event which was eliminated from a transaction by the new definition of transaction would not have passed as an event which could be part of a procedure anyway. If we can do this, then the results given in [Hewitt and Baker, 1975] for continuous functionals will still hold, since they are based on actors which behave like mathematical functions, and mathematical functions depend on procedures for their definition.

We have already characterized the events which were eliminated from transactions. Those which are neither request nor reply events can not be part of a procedure under the first restriction. Those request events whose corresponding reply events were not part of the transaction cannot be part of a procedure either, under the following reasoning. Assume the existence of a request event RQ which is a member of transaction(R), but whose reply RP is not. Then RQ is also a member of transaction(RQ), as is its reply, RP. Then transaction(R) and transaction(RQ) are not disjoint in that they both contain RQ, but there is no containment since RP is not an element of transaction(R) (therefore transaction(RQ) is not contained in transaction(R)), and since R--->RQ, R cannot be an element of transaction(RQ) (therefore transaction(R) is not contained in transaction(RQ)). Thus, no such transaction would pass as a procedure anyway.

Thus, even with the new improved definition of transaction, we can still show that if an actor behaves like a mathematical function, then it is the limit of a continuous functional in the sense of Scott. It remains to be seen if analagous results can be shown to hold true for order-dependent actors.

IV. Conclusions

We have uncovered two "bugs" in the [Hewitt and Baker, 1975] paper, one with the definition of "reply", and one with the definition of a "transaction". We proposed alternative definitions for both, and showed how these new definitions solved the discrepancies raised by the original definitions. Using the new definition of transaction, the *Law of Containment for Transactions* was proved, and the definitions of a procedure and a mathematical function were shown to hold true. Because these definitions held, we were able to maintain the result that if an actor behaves like a mathematical function, then it is the limit of a continuous functional in the sense of Scott.

V. Future Work

We have not yet discussed the uniqueness of replies, or indeed how multiple replies might affect the definition of a transaction. Although normally a request has only one reply, it is conceivable that an actor might have a behavior that causes multiple replies to be sent in response to some request.

VI. Acknowledgements

I wish to thank Carl Hewitt for many valuable discussions on transactions and actors. Bill Kornfeld and Roger Duffey acted as helpful sounding boards for some of my ideas, and encouraged my quest for the "perfect transaction".

VII. Bibliography

Hewitt, C. and Baker, H. Actors and Continuous Functionals. MIT LCS TR-194,December 1977

Hewitt, C. and Attardi, G. Proving Properties of Concurrent Programs Expressed as Behavioral Specifications. In preparation.

Hewitt, C. and Baker, H. Laws for Communicating Parallel Processes. MIT Artificial Intelligence Working Paper 1314A. December 1976.Invited paper at IFIP-77

Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages". AI Journal, V8, 1977. pp323-364.