Multiparty Interactions for Interprocess Communication and Synchronization

MICHAEL EVANGELIST, NISSIM FRANCEZ, AND SHMUÉL KATZ

Abstract—The essential properties are considered of a multiparty interaction construct which serves as a primitive for interprocess communication and synchronization in distributed programs. It is claimed that more general constructs, which violate the suggested properties, are inappropriate for abstraction, but should not be seen as a communication primitive, and that both facilities are needed. Several acceptability criteria are posed for multiparty interactions, and various possibilities for constructs satisfying these criteria are presented. These include introducing a new kind of nondeterminism within the assignments of an interaction, allowing restricted iteration within interactions, weakening the synchronization among the participants in an interaction, and varying the number of participants in order to provide a high-level treatment of fault tolerance.

Index Terms—Fault tolerance, horizontal nondeterminism, interprocess communication, multiparty interaction, quorum interaction, synchronization mechanism.

I. INTRODUCTION

In designing languages for distributed programming, there are several possible approaches to incorporating a construct which generalizes a point-to-point activity to a multiprocess activity. These approaches have in common that numerous processes will somehow “come together” to generate a temporary combined state, perform some actions on the combined state, and then separate to continue executing independently.

We strongly believe that this approach is promising and that future languages for distributed programming will provide more of these constructs, in addition to several existing proposals in this vein. The current prejudice in favor of binary message-passing constructs seems to us to reflect too closely certain architectures which are common in implementing physically distributed processes, but are not the only ones possible. In fact, these binary constructs are often not natural for describing transfers of information. They are simply too “low-level” and are more appropriate for an assembly language of distributed programming than for a high-level language. Thus, a better understanding of the functions and properties of multiparty constructs is an important and relevant research topic. We present here some steps in this direction, in an informal manner. Some existing formal studies are indicated where appropriate.

One purpose of a construct involving multiple processes is to create abstractions and modularize distributed programs, so that an abstraction can be defined only once but used repeatedly, possibly by varying process groups. The abstraction encapsulates representation details, including lower-level interprocess activity. Such an abstraction idea is captured by the script [18] or team [13] constructs, which have formal roles to which an actual process may enroll. While crucial for the development of distributed systems, this type of construct is not treated in this paper.

Rather, we stress that the above abstraction constructs encapsulate multiple activations of a lower-level operation for the actual synchronization and communication among subsets of roles. Here we concentrate on this communication primitive, which is generically called an interaction. Interactions may be in mutual conflict by having nondisjoint sets of participants. The issue of conflict resolution (how it is decided which eligible interaction will occur) is orthogonal to the question of the appropriate contents of an interaction or the criteria for eligibility and is not treated here. Conflict resolution is related to the notion of fairness and its appropriate interpretation in this context, as seen in [2], [3], and [6]. Also, we do not treat here the nature of the agents participating in interactions (formal roles, actual processes, naming conventions, etc.). In the sequel the term role is used generically for an arbitrary participant in a multiparty interaction.

One extreme approach to constructs for multiparty activity is seen in the joint actions of [5], [6], shared actions of [23], or the multiway rendezvous of [8]. There an arbitrary sequential program (including a nonterminating one) may be executed on the combined state during the interaction. However this violates our intuitive notion of a communication primitive where the full expressiveness needed for general protocols are left to the abstraction or encapsulation mechanisms.

At the other extreme, the simplest multiparty activities mentioned in the literature are the broadcast or more generally the multicast primitives (see, e.g., [7], [9], [19], [20], [24]), in which exactly one value is passed from one participant to all others (or a subset thereof) in the second
case). We refer to such primitives as directed interactions. Their characteristic is that the activity is directed in that the value is sent to one or more recipients, as in message-passing. Most of these constructs do not serve to synchronize the participants.

Neither joint actions nor broadcasts are appropriate for the level of abstraction needed in a multiparty interaction construct. Here we reserve the term "interaction" for activities defined on a combined state, where each participating role may both provide and use state components. Thus, this is an undirected activity, which will synchronize its participants. On the other hand, actions allowed within an interaction will be severely restricted.

To provide criteria for evaluating the appropriateness of various possibilities and to further justify considering the above constructs nonprimitive, we suggest several properties as characterizing a multiparty interaction that serves as an interprocess communication primitive:

1) Synchronization Upon Entry: The main effect of this requirement is to provide a consistent combined state for the interaction. We consider other alternatives in Section VI.

2) Split Bodies, Affecting Only Local States: By this we mean that each participant role in an interaction includes within its own program an interaction body which lists all changes to its local variables within the interaction. These changes may depend on the values of variables of other participants. The collection of relevant local bodies determines the effect of the entire interaction. This feature is particularly useful in that it facilitates the verification of programs using interactions by applying the cooperating proofs methodology, which has proven useful for many other communication primitives (see [17]).

3) Interprocess Access to Variables of Participants Only Within Interaction Bodies: Outside of interactions the roles can only use their local variables, and the usual state-dishappiness of distributed processes applies.

4) Access Only to the Initial Local States of All Participants: In other words, an interaction may be viewed as if the combined state is frozen to those values which existed at the synchronization point, until the end of the interaction, when all changes which were made take effect.

5) Bounded Duration: The number of steps needed to implement an interaction using lower-level operations should depend only on the number of participants and the number of values used, and not on the combined state in which it was started. This justifies using the number of interactions executed during a computation as a complexity measure, generalizing the more traditional message complexity.

These features seem necessary in order to view an interaction as a generalized communication primitive. This is because they allow naturally mapping the construct to known lower-level (and in particular, point-to-point) interprocess communication primitives, with a globally fixed number of communications, dependent only on the number of participant processes and the number of variables used within the local bodies, but not on the contents of the combined state itself. It becomes reasonable to consider the number of interactions executed as an indication of the time complexity of the programs which use them. In particular, property 4) is crucial, since it disallows general iteration within an interaction body. In more general constructs, such as the joint or shared actions mentioned above, this simple mapping is lost since an arbitrary sequential program on the combined state may take place after synchronization.

One may wonder why synchronous termination of an interaction (wherein its participants are jointly released) is not considered as a separate issue, e.g., as in [18]. The answer is that, even with full synchronization upon entry, as long as some local state is always assumed to exist between two consecutive executions of interactions, then synchronous termination of interactions is unnecessary. This is because the other properties ensure that any changes to the state made by a role are entirely local, and can be sensed by another role only by participation in a subsequent interaction. That latter interaction again requires synchronization of all participant roles upon entry, and thus by that time all of the participants must have finished their part in the previous interaction and completed any indicated local actions. There is no way to externally distinguish a process still active in a previous interaction body from one "resting" in the immediately following local state.

In Sections II–V, we deal with the structure of the allowed global actions within an interaction, rather than with the way the roles "come together." The question of the degree of synchronization (e.g., whether all the participants must initiate executing an interaction at the same time) is treated in Section VI, and varying the number of participants (needed for fault tolerance) is considered in Section VII.

II. SIMPLE INTERACTIONS

The above considerations suggest an unusual view of communication as "taking" the needed values rather than "sending" the appropriate values, as seen in common message-passing operations. Participation in an interaction means that the ("frozen") values of the variables of that process are momentarily available to other processes, and that the process may use variables of other participants to update its own variables. Thus an interaction can be seen as a temporary extension of the scope of each participant to provide read-only access to variables from other participants. Outside of interactions, a role has access only to its local variables. The primitive nature of an interaction is maintained by providing access only to the values of the variables when the interaction was begun.

The above approach to interprocess communication can also be seen as an implicit provision of needed values, as opposed to the explicit provision seen in message passing. Implicit provision has some advantages over the message passing view, in particular for modularity. Changing what is needed by one of the roles in an interaction does not require changing anything in the other roles: the changed
role will simply use different components of the implicitly
provided information. Even adding new participants can be
done without changing the other roles. In the message
passing view, of course, the other participants would have
to be modified to explicitly send the needed values.

The simplest form of an interaction (as seen in [4], [13])
is quite straightforward: each interaction has a name and
a collection of local bodies, at least one for each partici-
pant. At this stage, we assume that the collection of par-
ticipants is statically determined, and is explicit in the
program text. In our syntax, a local body of an interaction
in a role appears within brackets after the name of the
interaction, and consists of assignments to variables local
to the role. The right-hand side of an assignment is given
by means of an expression over the combined state of the
interaction (i.e., possibly referring to variables local to
other participating roles). Thus, a typical form of a local
body might look like \[ x_1 := e_1, \ldots, x_n := e_n ] \].

In general, a role may have more than one local body
of the same interaction, with different local updates, at
different points in its code, to be used on different exe-
cutions of the interaction. The empty body \([\{\}\) is fre-
cently used, indicating that the role with that body
merely needs to synchronize with the other participants
and/or make its local variables available to the other par-
ticipants.

Expression evaluation is assumed to be always termi-
nating with a well-defined value and not to have any side
effects. The expressions are taken from an underlying
(expression) language, not further specified here. We only
assume that the overhead of computing the expressions is
negligible compared to the overhead of synchronization
and transfer of the values.

A role is said to ready an interaction if that role has
reached a point at which some local body of the interac-
tion is a possible continuation. An interaction is enabled
if all its participants have readied it. As in the accept
statement of \(\text{Ada} [1] \), or \(\text{CSP} [21] \) input/output, an inter-
action body can either be a statement or serve as a guard
of a nondeterministic selection statement. One alternative
in a selection thus has the form

\[
\text{locbool; interactname [interactbody] \rightarrow statement.}
\]

This means that if the local boolean condition \(\text{locbool}
\) holds in the current local state, and the interaction \(\text{inter-
actname} \) is selected for execution, then the interaction
body \(\text{interactbody} \) will be executed in parallel to the local
bodies of the other participants. The interaction is then
followed in this role by the execution of the statement
to the right of the arrow.

Here and in the continuation we will assume for sim-
plicity that all variable names are unique in the program,
or can be made unique by appending the name of the pro-
cess to the variable name where confusion could other-
wise result. An abstract language \(\text{IP} \) based on this sim-
plest view of interactions is presented in [3], [17] and
serves as a basis for more formal studies.

There are two closely related ways of operationally de-
scribing the effect of such an interaction. According to the
first, each participating role copies to special temporary
local variables all regular nonlocal variables whose values
are needed. Each participant is released after all copying
activities in which it is involved have been completed.
Then, after the interaction, each role can locally compute
needed expressions, and perform the needed assignments
to the regular, nontemporary variables. In the second ap-
proach, each participant first copies its local variables to
special temporary locations, and then copies the variables
it needs from the temporary locations of the other partic-
pants, computes needed expressions and assigns the re-
results directly into its regular variables. Either case greatly
increases the size of the local state of each role (in the
worst case requiring variables which replicate the entire
"regular" global state within each role). Moreover, the
program may be difficult to follow and correctness rea-
soning may be complicated, since copying is separated
from actual use (computing and updating).

An interaction is therefore defined by the collection of
its local bodies, and their distribution among the roles. It
is not strictly necessary to assume a specific declaration
point for an interaction, external to the participants. How-
ever, such a declaration, which would merely list the
names of the participant roles, is valuable in understand-
ing the scope of an interaction, and in allowing static
checks that each intended participant contains at least one
local body for the interaction. The form of a declaration is

\[
\text{interaction interactname with participant-list.}
\]

Some of the extensions suggested later assume such a de-
claration and add to it features which are independent of
the local bodies.

Below we show that the nature of an interaction can be
extended while preserving the features which justify call-
ing the interaction a communication primitive. We de-
scribe in stages several such generalizations.

III. CONDITIONAL ASSIGNMENTS

The basic idea of this modest generalization is to allow
the assignments within a local interaction body to depend
on (possibly global) conditions over the combined state.
The goal is to avoid unnatural copying and breaking up
of operations to several stages. If \(\text{assignment-set} \) denotes
a collection of assignments to local variables, separated
by commas, the form of an interaction body within a role
\(R \) becomes

\[
in\{b_1 \rightarrow \{\text{assignment-set}_1\}, \ldots, b_n \rightarrow \{\text{assignment-set}_n\}\}.
\]

Both \(b_i \) and the right-hand sides of the assignments may
refer to variables local to the states of other participant
roles, while the left-hand sides of the assignments are
variables local to \(R \), the role having that body. When only
one assignment is in the assignment-set, the brackets can
be omitted. Below, several programming/design situa-
tions are described where this generalization seems convenient.

1) Suppose $P$ and $Q$ are roles which are auditing different activities of an encapsulation representing a bank. One of them might keep track of the number of days the total balance of the bank is positive, while the other keeps track of negative (or zero) balances. If the bank’s various accounts are maintained by different roles $R_i$, $1 \leq i \leq m$, each in a variable `account`, then in a periodically performed interaction `balance`, where each $R_i$ has either `balance` or does some locally needed computation in the interaction, the body of $P$ might contain

\[
\text{balance} \left[ \left( \sum_{i=1}^{m} \text{account}_i \right) > 0 \rightarrow \text{poscount} := \text{poscount} + 1 \right],
\]

while the body of $Q$ would have

\[
\text{balance} \left[ \left( \sum_{i=1}^{m} \text{account}_i \right) \leq 0 \rightarrow \text{negcount} := \text{negcount} + 1 \right].
\]

Using the conditionals in this example allows doing the required operations in one interaction, instead of separately checking the account balance and the updating into two interactions. Whether conditionals add expressive power is discussed after the other examples. The fact that the sum of the accounts is computed twice, once in $P$ and once in $Q$, is still problematic, and is treated in Section V.

2) Consider a token-passing distributed system with a single token and a single variable $a_i$ for each role representing the dynamically changing priority of the role. At this point, we assume that priorities are invariantly pairwise distinct. Each role has a boolean variable $t_i$ which will be true only if that process is holding the abstract token (and only one process at a time can have $t_i$ true). Within a token-passing interaction $tp$, the token should be passed to the role with the highest priority. (For convenience, we assume that the process with the highest priority is already tokens.) In the extended notation, each role would have the following interaction body

\[
\text{tp} \vdash t_i \land (a_i = \max_{i=1,n} a_j) \rightarrow t_i := \text{true},
\]

\[
t_i \rightarrow t_i := \text{false}.
\]

Note that roles which have neither the token nor the maximal priority value do nothing after evaluating the tests.

3) Suppose the maximum operator $\max$ is not present in the underlying expression language. In order to assign to a local variable $z$ the maximum of the variables $x$ and $y$, each in different roles, the natural interaction body in the role holding $z$ would consist of

\[
\max \{ y > x \rightarrow z := y, x \geq y \rightarrow z := x \}.
\]

As noted above, a natural question is whether adding the feature of conditional assignments enhances the expressive power of interactions. It may be argued that the conditional assignments within interactions can be expressed by two consecutive simple interaction bodies: one to determine the truth of the global conditions, and then another to make the necessary changes. For example, consider an interaction with a single conditional assignment in each of two roles: $R_1$ containing $\text{inter} \{ b_1 \rightarrow x_1 := e_1 \}$, and a role $R_2$ containing $\text{inter} \{ b_2 \rightarrow x_2 := e_2 \}$.

Following the idea above, this could be expressed by having in $R_1$:

\[
\text{check} \{ \text{cond}_1 := b_1 \rightarrow \text{doint} \} \text{ if } \text{cond}_1 \text{ then } x_1 := e_1 \}
\]

and in $R_2$ having

\[
\text{check} \{ \text{cond}_2 := b_2 \rightarrow \text{doint} \} \text{ if } \text{cond}_2 \text{ then } x_2 := e_2 \}
\]

Several comments on this simulation are in order:

1) If in the original interaction $\text{inter}$ appeared as a guard, and was preceded by local conditions, then those same conditions should be inserted before $\text{check}$. The statement to the right of the arrow after the guard would be executed sequentially, after the if-then-else statement of the simulation.

2) Note that the interaction $\text{check}$ is executed under exactly the same conditions as those for the conditional interaction $\text{inter}$. Then both participants can only execute the interaction $\text{doint}$ once. The execution of this second interaction is guaranteed only by an additional fairness condition [16]. In this case weak fairness is sufficient, since neither of the participant roles can do anything else and the interaction is continuously enabled. For a comprehensive discussion of fairness notions for multiparty interactions see [3].

3) The above expressibility claim is correct only under an additional restriction, namely that in case of formal roles, only a single activation of a role is permitted. In other words, no concurrent activations of the same role are permitted. Otherwise, the above simulation would be incorrect: the two interactions could be with different sets of activations of the roles, breaking the atomicity of the original conditional assignment. In case such multiple enrollments are permitted, as in Raddle [4], extra means of exclusion need to be added to the simulation, equivalent to a semaphore. This is achieved by yet another interaction with a semaphore role.

4) A generalization of this mode of expression to the more common situation with multiple assignments with different conditions for each, and of more than two participant roles, would become untenably complex. If a second conditional assignment were added to the body of $R_1$, then the four possible combinations of boolean results of the checking of the conditions would have to be considered, each with a different interaction body for $\text{doint}$.

A second approach to simulation is possible, using the copying idea in the simplest version of an interaction. That is, each role $R_i$ merely copies all nonlocal variables which
would have appeared in a conditional interaction to local variables. Then, after the interaction, the role can locally compute the conditions and the expressions, and perform the needed assignments. The disadvantages of this approach are only increased in that now a low-level encoding of the inspection of the conditions and choosing among them would make this even less readable than already indicated. Moreover, extraneous copying might be done for the variables from expressions which are not ultimately computed because the condition evaluates to false.

IV. HORIZONTAL NONDETERMINISM

With the addition of conditionals, local interaction bodies are made up of guarded commands with sets of assignments as their right-hand sides, and with (global) boolean expressions acting as guards. In all of the above examples, the conditions affecting assignments to the same local variable were mutually exclusive and exhaustive. However, it is now well understood from the sequential programming context that such a restriction is both arbitrary and unnecessary [11]. Removing it leads to what we call vertical nondeterminism within an interaction body. That is, within a local interaction body the conditions may be true for more than one assignment to the same variable, and exactly one of the assignments with a true condition will be executed for each variable. If no conditional appears, the guard true is assumed. Thus the two conditional assignments \( y \geq z \rightarrow x := x + y \) and \( z \geq y \rightarrow x := x + z \) may appear on the same interaction body, both naturally referring to their right-hand sides to the initial value of \( x \), with the interpretation that when both guards are true (i.e., \( y = z \)) exactly one of the assignments will be chosen to be executed. In case no guard of an assignment to a variable \( x \) is true, then no change will be made.

In addition, we suggest incorporating what we term horizontal nondeterminism in interactions. This involves choosing one of a set of assignments from different interaction bodies in different roles within the same interaction. Consider the following situation: a collection of \( n + 3 \) roles is designed to implement the abstract data type \( SET(n) \), of a set with up to \( n \) elements. The collection has a role for each of the standard set operations insert, delete, and member. These roles interact with the implementing roles \( R_i \), \( 1 \leq i \leq n \). Each implementing role has a local variable \( a_i \) which may hold one element of the set, and a boolean variable empty, indicating whether \( a_i \) currently holds a member of the set. The role insert(\( x \)) has the interaction body ins[1], the role delete(\( x \)) has del[1], and the member(\( x \), haveit) role includes mem[haveit := \( \forall_{i=1,n} (\neg \text{empty}_i \Rightarrow a_i \neq x) \) \( \land \text{empty}_i \)]. The intention is that an implementing role may participate in the ins interaction and copy \( x \) into \( a_i \) if \( x \) is not already in the set (expressed by the global condition \( \forall_{i=1,n} (\neg \text{empty}_i \Rightarrow a_i \neq x) \)) and \( \text{empty}_i := \text{true} \) holds. Clearly, several \( R_i \)'s may satisfy the conditions, and it does not matter which one of them performs the assignment, but exactly one of them should do so, in order to prevent replications in the representation of the set.

Similarly, in a collection of roles intended to represent the elevator system of a building, e.g., the lift example presented in [14], there are many elevator roles. For each external request for service, each elevator has a way to determine its own costs for providing that service. The elevators interact to elect the one with a minimal service cost. If the costs of service are not distinct for each elevator, it really does not matter which one provides the service. Among those of equal minimal cost, exactly one should assign its mine variable to true in the election interaction, thereby representing the fact that it is to satisfy the current request.

We believe that this idea of horizontal nondeterminism is a new suggestion. Usually, the only nondeterminism across processes is conceived as originating from speed differences. This arises in semantic definitions when nondeterministic interleaving of independent processes (with or without fairness) is taken to represent concurrency. Indeed, our new proposal makes sense only in synchronized multiparty activities, where a combined state is temporally formed and common knowledge among the participants is achievable.

In order to express this idea syntactically, we need a way to "color" various assignments in an interaction, from different local interaction bodies, and interpret this so that only one assignment of each color will be performed when the interaction is executed (leaving the choice to the implementation). This is most simply expressed syntactically by labeling conditional assignments in interaction bodies. The form of a clause in a local body becomes \( L : b \rightarrow \{ \text{assignment-set} \} \). The scope of a label is the collection of bodies of a single interaction. The interpretation is that only one of the assignment-sets with a true condition and the same label will be executed during each execution of the interaction. If none of the conditions with the same label evaluate to true during an execution of an interaction, then no assignment will occur for that label. The default if no label appears is that a unique label is assumed for all assignments to the same local variable.

The insert interaction ins then has the definition

interaction ins with insert, \( R_i \), \( 1 \leq i \leq n \).

In each implementing role \( R_i \), the body of the interaction is

\[
\text{ins} : \left( \left( \land_{i=1,n} (\neg \text{empty}_i \Rightarrow a_i \neq x) \right) \land \text{empty}_i \right) \\
\rightarrow \{ a_i := x, \text{empty}_i := \text{false} \}.
\]

The first conjunct of the guard expresses that the value \( x \) is not already in the set, and the second that this role is available for inserting. Since all implementing roles have the same label \( L \) for the interaction, only one role with a true guard will execute the assignments. The insert role
merely provides the needed value for insertion by including the body \text{ins}[i].

The elevator system could express the election interaction by having the body

\[
\text{election} \left[ N : \left( \text{cost}_i = \min \text{cost}_j \right) \rightarrow \text{mine} : = \text{true} \right]
\]

in each participant elevator \( P_i \).

An implementation of the horizontal nondeterminism construct should solve (at run time) the following problem, which we call the multicolored leader-election problem: for a graph with an assignment of a finite, nonempty set of "colors" to each of its vertices, find a unique distinguished vertex (a "leader") for each color present in the graph.

For our application, each color is an instance of horizontal nondeterminism in the interaction, and the leader is the role selected to execute a statement of that color. This problem can be solved under a variety of assumptions about the graph (cycle, full graph), about the color assignment and about the target-function to be optimized under various complexity measures. For example, one could consider finding the minimum number of leaders covering together all colors. This would mean, for the application at hand, that the smallest number of interaction participants are active, each doing as much as possible. On the other hand, one could consider finding the maximal number of such leaders, meaning that the largest number of participants are active, each doing the smallest possible task. At this stage, there does not seem to be a clear preference as to which are the more practical and interesting variants of the theoretical problem stated above. More might be learned from actual implementation efforts.

The generalizations suggested above maintain all properties we consider essential. The correctness reasoning is only slightly more complicated in that now some of the assignments might not be executed, either because the condition does not hold, or because they are part of a nondeterministic choice. However, similar reasoning has to be carried out in any simulation using the simplest definition of interaction.

As previously, for some cases it is possible to express the new structure by means of the previous versions. In the set example, the \text{insert}(x) operation could be done by separating the test of membership of the argument \( x \) in the set, and its actual inclusion. The test can be done in one \((n + 1)\)-party interaction (called \text{checkmem} in the code below), while the actual copying, which we did non-deterministically, uses a collection of binary interactions \text{copy}, between any one of the empty implementation roles with the insert role. All of the empty roles will compete to carry out "their own" interaction, so the conflict resolution mechanism can be used to achieve the nondeterministic choice. The \text{insert} role can guarantee that some such interaction will occur whenever there is at least one empty role, and the element has not yet been inserted.

Note, however, that it is necessary to have all the roles participate in another \((n + 1)\)-party interaction (called \text{go-on} below) in order to announce that the insertion has been completed, and "free" them for other activities.

The declarations of the interactions are then

\[
\text{interaction checkmem with insert, } R_i, 1 \leq i \leq n
\]

\[
\text{interaction go-on with insert, } R_i, 1 \leq i \leq n
\]

for \( 1 \leq i \leq n \). interaction \text{copy}, with insert, \( R_i \)

The \text{insert}(x) role would appear as:

\[
\text{checkmem}[c := \bigwedge_{j=1,n} (\neg \text{empty}_j \Rightarrow a_j \neq x) \land \bigvee_{j=1,n} \text{empty}_j];
\]

\[
\bigwedge_{j=1,n} c; \text{copy},[i] \rightarrow \text{go-on}[]
\]

\[
\neg c; \text{go-on}[] \rightarrow \text{skip}
\]

The role \( R_i \), then consists of:

\[
\text{checkmem}[c];
\]

\[
\text{empty}; \text{copy},[i]; a_i := x \rightarrow \text{go-on}[]
\]

\[
\text{go-on}[] \rightarrow \text{skip}
\]

In this implementation an insert operation when all roles already have values will act like a \text{skip} operation. A minor modification (left to the reader) would return an explicit error indication. In a more general context with multiple instances of horizontal nondeterminism among different subsets of participants, it is not obvious how to similarly exploit the conflict resolution mechanism. This is because there need not be a single participant in which it is natural to concentrate the alternative choice among the \text{copy}, interactions, as was done in \text{insert}, above.

V. RESTRICTED ITERATION AND LOCAL INTERACTION VARIABLES

Restricting our generalized communication construct to primitive operations is intended to encourage modular program structure. That is, a natural modularity is created by limiting program behavior within the communication primitive and, therefore, forcing other functionality into local computations on the one hand, and more global multiparty abstractions on the other hand. Constraining communication actions also helps ensure that comparing the number of communications in two programs is meaningful. If arbitrary actions were allowed, the units of comparison could vary too much to be significant. In the following proposals for a generalized communication primitive, we extend the interaction construct to permit a richer communication language without violating these goals.

So far every variable appearing in an interaction was local to some participant role. A consistent and useful extension of this idea may be formulated as follows:

Let \( in \) be an interaction in which roles \( R_1, R_2, \cdots , R_n \) participate. Associated with each role, \( R_i \), is a set, \( X_i \), of local variables used during some execution of \( in \). We sup-

Authorized licensed use limited to: Kansas State University. Downloaded on February 12, 2010 at 10:32 from IEEE Xplore. Restrictions apply.
pose that every time in is committed to execution, a set 
\( W = \{ w_1, w_2, \ldots, w_m \}, 0 \leq m \), of interaction vari-
ables is created. The \( w_i \) have undefined initial values and can
be referenced only within the instance of interaction execution for which they were created. If \( X \) is the union of all the \( X_i \), then we insist that \( X \) and \( W \) be disjoint, to prevent ambiguous reference.

In this formulation, the \( w_i \) are used only for temporary storage in a synchronized computation. In order to express this possibility, the computation is viewed as taking place in two stages:

1) In the first stage, only variables from \( W \) can appear
on the left side of assignment statements; only variables
from \( X \) can appear on the right side. The assignments are
executed in parallel.

2) In the second stage, only variables from \( X \) can appear
on the left side; variables from both \( X \) and \( W \) can
appear on the right side. Again, the assignments are
executed in parallel. Such an extension is in accord with
conventional approaches in language design, where units
of modularity become also units of scope, facilitating both
information hiding and efficient storage utilization.

Since the first stage is not associated with any one body,
it is natural to assume that an interaction will be declared,
outside of any of the participants (as suggested in Section
II). A typical syntax for such a declaration could then be:

```
interaction name with list of participant role names
var list of interaction variables
first-stage body
```

The second stage is expressed as before, in interaction
codes local to the participants. In an implementation, the
first stage can be seen as a “hidden participant” which
computes the auxiliary values, and then provides them to
the other participants before they compute their interaction
bodies.

We demonstrate three applications of interaction vari-
ables:

1) Consider the distributed solution to the lift problem
[14] described earlier. In this program, each lift role com-
putes its cost for servicing a request, and then the mini-
mum of these costs is computed by all roles in an n-party
interaction among the n lifts. Since costs are unique, the
role whose cost matches the minimum knows it has been
elected to service the request. Using interaction variables,
this minimum may be computed once and stored in an
interaction variable. This example is in fact a practical
application of the problem of leader election in a distrib-
uted system. The leader is elected by comparing the value
of the interaction variable with each local cost variable.
The code for the declaration and interaction bodies is

```
interaction election with \( P_i \), 1 \leq i \leq n
var leastcost: real

election [leastcost := \( \min_{j=1,n} \text{cost}_j \)]

while in each \( P_i \) we have:

election [L: cost, = leastcost \rightarrow min_e := true].
```

2) In the top-level design of the electronic-funds trans-
fer system in [15], each role has a local variable okay that
is computed by each role in an interaction among the
point-of-sale terminal, the customer bank, and the
merchant bank. The variables okay will be true after this
interaction if, and only if, the customer’s bank account is
valid and contains sufficient funds for the transaction.
Here, the needed value of okay would be computed only
once, as an interaction variable, and then used in the test
for whether the transaction should proceed.

3) The insert operation of the set implementation pre-
sented in the previous section is most naturally expressed
as a two-level computation. The declaration and the in-
teraction bodies are

```
interaction ins with insert, \( R_i \), 1 \leq i \leq n
var c: boolean

ins \[c := \land_{j=1,n} (\neg \text{empty}_j \Rightarrow a_j \neq x)\]

where insert again simply has the body ins [], while in each \( R_i \) we have:

ins [L: empty, \land c \rightarrow \{a_i := x; \text{empty}_i := false\}].
```

In these examples each role needs the result of the same
computation as an expression within the interaction bod-
ies. Since the expression is based on information from all
roles involved, the easiest way to communicate the result
in the original version of each example was to have each
role compute the same value independently. Interaction
variables, by allowing a preliminary computation using
temporary variables, permit unique computations of this
value. This extension does not allow unbounded computa-
tions. Thus, we may still meaningfully compare the
number of interactions in two programs. Although opti-
mizations are possible, in a straightforward implementa-
tion of interactions using (binary) message passing, when
each role needs the same function dependent on values
from all other roles, using an interaction variable requires
sending only \( O(2n) \) values, while without such a variable
\( O(n^2) \) values would be sent.

It should be clear that we can further extend the lan-
guage of communication by allowing fixed iterations in
an interaction. If the number of iterations does not depend
on local states, then interaction complexity can still differ
only by a constant. Function calls that do not violate this
principle (and, of course, do not contain interactions) may
also appear.

VI. WEAKENING SYNCHRONY

So far, full synchronization of all participant roles has
been assumed at the beginning of an interaction. There
are various possibilities for relaxing this assumption. In
this view it is easiest to consider an interaction as a three-
stage operation of registration, obtaining values, and up-
dating (assignments). Registration means committing to
the interaction and providing the relevant current values
of the role’s local state, which will be used by the other
participants in the interaction. As already indicated, even if the registration is synchronized, subsequent stages need not be, since no new interaction can begin which involves the role until it has completed its part of the previous interaction.

In considering relaxations of synchronization at the beginning of an interaction (the registration stage), one reasonable view (which is adopted throughout this section) is to allow possibilities which are "equivalent" to a full synchronization. That is, for every execution \( E \) under the relaxed restriction, there should be an execution in which all roles begin the interaction at the same time and which differs from \( E \) only by having interchanged independent events. In general, this will not be possible when the interactions act as guards for nondeterministic choices, where the guard is "passed" only when other participants are already committed to the interaction, and an associated right-hand side is then executed. For example, if a role \( P \) can choose between \( \text{in}_1[x := 1] \) and \( \text{in}_2[x := 0] \) and a role \( R \) between \( \text{in}_3[y := 1] \) and \( \text{in}_4[y := 0] \), then the synchronized version will always terminate with \( x = y \). However, if \( P \) can commit to \( \text{in}_1 \) while \( R \) commits to \( \text{in}_2 \), the computation will either terminate with \( x \neq y \) or be deadlocked, depending on whether or not interactions must be eventually completed by all participant roles. In either case, the computations are not equivalent to the original. Thus below it is assumed that interactions do not act as guards.

This means that we should be able to consider the interaction and following local action as a layer of computation [12] equivalent to a synchronized version. Such a view is also advocated in a logical framework in [22], where an equivalence relation between computations with a synchronization and those without is examined. For example, a requirement could be: "Only if all roles which participated in an interaction \( \text{in}_1 \) involving \( P \) have completed registering for it, and \( P \) was provided with current values of the variables it needed in \( \text{in}_1 \), may \( P \) register for a new interaction \( \text{in}_2 \)."

In other words, a role may register for an interaction without all other participants being available, and under certain circumstances even complete the interaction and subsequent local computation, but may not begin the next interaction until all of the participants have at least begun the previous one. Such a restriction ensures that the equivalence indicated above will indeed hold, no matter how much additional synchronization is or is not required. Without such a restriction, even disallowing interactions as guards for nondeterministic choices, execution sequences may be obtained which are inconsistent with those which could have occurred when full synchronization was required. In particular, for situations in which the original computation could deadlock, a non-synchronized one might not, and vice versa.

In addition to the restriction above, it is possible to maintain partial synchrony, among those participant roles which actually need to transfer values in order to allow the necessary computations within the interaction. A dependency graph among the participant roles can be constructed for each interaction where \( P \) is connected to \( Q \) if a value of one is used by the other (the direction of transfer is irrelevant here). Then each connected component of the graph would synchronize when the interaction is executed.

As an illustration of this idea, suppose we have roles \( P_1, 1 \leq i \leq 6 \), each with a local variable \( x_i \). The bodies of the interaction contain the assignments:

\[
\begin{align*}
x_1 & := f(x_1, x_3, x_4) \\
x_3 & := g(x_2, x_5) \\
x_5 & := h(x_3, x_6)
\end{align*}
\]

and the variables \( x_2, x_4, \) and \( x_6 \) are not modified. In this case the group of roles \( P_1, P_2, P_3, P_4 \) would synchronize in computing \( x_1 \) and \( x_3 \), while \( P_5, P_6 \) synchronize in computing \( x_5 \).

It is also possible to consider only direct dependencies and then allow several synchronizations per role within the same interaction. In the example above, the group \( P_1, P_3, P_4 \) would synchronize in performing the first assignment, \( P_2 \) for the second, and \( P_5, P_6 \) for the third. A role then finishes participation in the interaction when all of the synchronizations in which it must participate have been completed.

VII. WEAKENING PARTICIPATION FOR FAULT TOLERANCE

In some cases, we wish to provide for the possibility that multiple parties may interact, but we do not insist that the same group of roles must always participate. If the roles are completely distinct, it is reasonable to denote particular subsets which are sufficient to allow an interaction to proceed. This idea is related to the notion of a critical set of participants as stated in [18]. A common special case involves parametric roles, where the code is symmetric, and then merely specifying the required number of such roles is sufficient to describe when the interaction may proceed. As long as some specified number of roles are ready to interact, in these cases, the computation may sensibly proceed. For simplicity, we consider this special case. An example of this situation is the distributed solution to the lift problem, previously discussed. Instead of requiring an interaction among all \( n \) lifts to determine which should service the current request, we could say that if any lifts are available, they should participate in the interaction. This formulation of joint communication allows us to express an important kind of fault tolerance, since the roles not participating may represent failed processes. Alternatively, nonparticipating roles may simply be involved in other computations and their presence not required if a threshold has been reached. In the lift problem, for example, one of the lifts may currently be in dedicated use and not available for general requests.

We call this idea quorum interactions and formulate it initially to support fault tolerance only. We assume that process failure can be detected. An implementation of this formulation of quorum interactions must determine that at
least \( k \) processes have not failed. (Failed processes can never re-enter the computation.) If the quorum threshold has been met, then we apply the usual enablement criteria. In this view, a failed process can be seen by definition as being at a control point for every action in which it participates.

Suppose \( \text{in} \) is an interaction among \( n > 1 \) parties. Then we may state that \( \text{in} \) can execute if \( r \) is the number of parties that have not failed and \( r \geq k \), \( 1 \leq k \leq n \), where the constant \( k \) is the quorum threshold, and the \( r \) parties are each at a control point where \( \text{in} \) is a choice and all are ready to participate. The interaction-definition section must specify default values for all local variables that appear in the computation. This is particularly useful in the context where the computed expressions are constructed by means of a commutative and associative operation with a unit (i.e., neutral) value, such as indexed sum.

By putting \( k = 1 \), we can use quorum interactions to make the distributed lift program tolerant to process failure. However, this formulation will not allow us to make provision for roles that merely wish to ignore a computational path, and therefore an interaction, for some period—as in the example above where a lift is not available. We refer to such a role as participating in a dedicated computation. In this case, the roles not participating in an interaction cannot be viewed as being at a choice point for that interaction. Since these roles are involved in a dedicated computation, waiting for them would lock the system until they finish.

To model such dedicated behavior with quorum interactions, we use a special boolean switch, called an \( \text{in-switch} \), \( \text{B}_n \), that is associated with the interaction, \( \text{in} \), and a process \( P \) in which \( \text{in} \) appears. \( \text{B}_n \) is declared in a declaration section for \( \text{in} \). When \( \text{B}_n \) is true, if \( P \) has not failed, it must participate in all executions of the interaction \( \text{in} \) within its body. In this case, the semantics is the same as in the previous formulation of quorum interactions for fault-tolerance. When \( \text{B}_n \) is false, \( P \) cannot participate in any execution of \( \text{in} \). Thus, \( P \) must either have alternatives to such participation or it must be the case that \( \text{B}_n \) cannot be false when an instance of \( \text{in} \) not having an alternative is encountered. When \( \text{B}_n \) is false, \( \text{in} \) takes data values from the defaults in the declaration section. If the quorum threshold cannot be met, because too many processes have false \( \text{in-switches} \), then \( \text{in} \) cannot be executed. Those processes with readied instances of \( \text{in} \) must either choose another action, if possible, or wait until a quorum is ready, if not. This idea allows dynamically changing the critical set of processes needed for the interaction to occur.

Such situations can be expressed by using an ordinary boolean and having \( \text{in} \) as a choice at each stage of the dedicated computation of \( P \). Here, \( P \) would merely synchronize with the other parties to the interaction, provide only the default data values, and then continue its dedicated computation. The advantage in using an \( \text{in-switch} \), as with most of the extensions to interaction semantics described in this paper, is its greater convenience and conceptual clarity over the simulation.

This idea is easily extended to a more general model for quorum interactions. Let \( S = \{ P_1, P_2, \ldots, P_n \} \) be a nonempty set of processes, and let \( \Pi = \{ \Pi_1, \Pi_2, \ldots, \Pi_r \} \) be a partition of \( S \). We suppose that \( \Pi_i = \{ P_{i1}, P_{i2}, \ldots, P_{in_i} \} \). Associate with each \( \Pi_i \) a quorum threshold, \( k_i \). Versions of the quorum interaction can be defined, as before, in terms of the fault-tolerance model or in terms of the dedicated computation model. For fault-tolerance: let \( S_i = \{ P_{i1}, P_{i2}, \ldots, P_{in_i} \} \) be the nonfailed processes. Of course, \( S_i \subseteq S = \bigcup_{i=1}^r \Pi_i \). If \( S \) is the process set for \( \text{in} \), then \( \text{in} \) may execute if every process in \( S \) has either failed or readied \( \text{in} \), and for each \( i \), at least \( k_i \) processes in \( \Pi_i \) have not failed. The formulation for the dedicated computation model extends the fault-tolerance formulation in the same way as before.

With this more general approach, we can model distributed computations where \( r \) different types of resources must be present in varying amounts. We give two applications of this model.

**Example: Control of Sensors**

Suppose \( n \) sensor roles repeatedly compute values, which are to be used by a control role \( M \) to provide an updated view of the environment. In order that the updated view be relevant, at least \( n/2 \) sensors must be available with values, while the others may be involved in internal computations, and not have a stable value available. A generalized quorum interaction among the \( n + 1 \) roles can be used to repeatedly update \( M \). Each sensor role \( T_i \) initially sets \( \text{B}_n \), \( \Pi_i \), to false. When \( T_i \) has a valid value and is available to participate in an update, it sets \( \text{B}_n \), \( \Pi_i \), to true, participates in the update interaction \( \text{up} \), and then resets the boolean to false. The partition is \( \Pi = \{ \{ M \}, \{ T_1, T_2, \ldots, T_r \} \} \), and the quorum thresholds are \( 1 \) and \( n/2 \) for the memory role and the sensor roles, respectively.

**Example: Readers-Writers**

In this version of the well-known readers-writers problem \([10]\), \( m \) reader and \( n \) writer roles access a common memory. Readers can share access with any number of other readers; no process can share access with a writer. Without quorum interactions, reader access must be specified by a separate interaction for each combination of the \( m \) readers as a choice. With the extension given above, we can use a single generalized quorum interaction (in the dedicated computation model) with two subsets: the memory role and the reader roles. The quorum thresholds are both \( 1 \). All readers with true interaction switches will participate in the interaction and be granted simultaneous access.

**VIII. Conclusions**

We have considered the nature of multiparty activities, dividing them into abstractions (or encapsulations) on the one hand and primitive interprocess communication constructs on the other hand. We believe that this distinction is important, and both aspects should be included in languages for distributed computing, but be separated, as
each one serves a different purpose. Their clear distinction enhances program readability and verifiability and leads to more efficient implementations.

We concentrated on the latter aspect of multiparty activities, namely as communication primitives called interactions. We proposed a general view of this function, and within it delineated the extent of stretching the contents of interaction bodies, the degree of synchrony, and conditions for activation. The main innovation is in the notion of horizontal nondeterminism, which specifies alternative actions across processes, and in considering various options for synchrony and fault tolerance.

ACKNOWLEDGMENT

We thank I. Forman for valuable discussions on the subject. We also thank R. Miller for suggesting the name quorum interaction and some of the applications. The work reported here was initiated during a visit of the last two authors at MCC, Austin, TX.

REFERENCES


Michael Evangelist received the Ph.D. degree in computer science from Northwestern University, Evanston, IL. He taught at Colgate from 1978 to 1982 and was a member of Technical Staff at Bell Labs from 1982 to 1985. He joined the Software Technology Program at MCC in 1985 and currently manages the Distributed Computing Group. His research interests include distributed computing, computational complexity theory, and software design. Dr. Evangelist is a member of the IEEE Computer Society and the Association for Computing Machinery.

Nissim Francez received the B.A. degree in mathematics and philosophy from the Hebrew University in Jerusalem, and the M.Sc. and Ph.D. degrees in computer science from the Weizmann Institute of Science, Rehovot, Israel, in 1976-1977 he spent a postdoctoral year at Queen’s University, Belfast, where he was introduced by C.A.R. Hoare to CSP. During 1977-1978 he was an Assistant Professor at the University of Southern California, Los Angeles. Since 1978 he has been with the Department of Computer Science at the Technion. During 1982-1983 he was on a sabbatical leave at IBM Thomas J. Watson Research Center. He has been a consultant for MCC’s software technology program, working on multiparty activities in distributed systems. He had summer appointments at Harvard University, IBM T. J. Watson Research Center, Utrecht University, CWI (Amsterdam), and at MCC. He also served on several program committees. His research interests include program verification and the semantics of programming languages, mainly for concurrent and distributed programming. He is also interested in log programming and recursive query evaluation and in compiler construction. He is the author of a book, Fairness.

Shmuel Katz received the B.A. degree in mathematics and English literature from the University of California, Los Angeles, and the M.Sc. and Ph.D. degrees in computer science from the Weizmann Institute in Rehovot, Israel. From 1976 to 1981 he was a researcher at the IBM Israel Scientific Center. Presently, he is a Senior Lecturer in the Department of Computer Science at the Technion in Haifa, Israel. During 1977-1978 he visited the University of California, Berkeley, and in 1984-1985 he was at the University of Texas at Austin. He has been a consultant and visitor at the MCC Software Technology Program, and in 1988-1989 he was a visiting scientist at the IBM Thomas J. Watson Research Center. His research interests include the methodology of programming, specification methods, program verification and semantics, distributed programming, data structures, and programming languages.