Language Constructs for Distributed Systems

by

Shmuel Katz, Ira Forman, and Michael Evangelist

MCC--Software Technology Program

Computer Science Department, The Technion

Abstract: Three language constructs appropriate for the design and programming of distributed systems are described. The team abstracts repeated interprocess activities and isolates processes to the context where they are needed. The interaction provides multiparty synchronization and communication. The superimposition defines modules that augment existing teams. Partial correctness proof rules and criteria for correctness-preserving transformations are given. These allow replacing one construct by others during design in order to reduce synchronization and extraneous communication.

Key words and phrases: design language, methodology of distributed programming, multiparty interaction, team, superimposition
1. Introduction

Designing correct distributed systems in a high-level language using a consistent programming methodology has proven to be an elusive goal. As a step towards this goal, three language constructs—teams, multiparty interactions, and superimpositions—are presented. The constructs are intended both for the programming and the top-down development of distributed systems. This means that both the executable code and the process by which that code is developed must be recorded and facilitated.

A design methodology for which the suggested constructs are appropriate emphasizes modularization and separation of concerns. These are essential whenever complex systems are considered. In addition, particular forms of refinement for distributed systems are shown to be a natural outcome of replacing one of the constructs by others during the design stage. The constructs encourage recording an evolving system design and maintaining generality that avoids overcommitment at too early a stage of design, while not abandoning a precise executable language. Note however, that since the design language is executable, a separate nonexecutable specification language will also be a necessary component in our methodology. This will be explained in Section 5.

The three constructs considered are orthogonal, and each provide a different type of abstraction mechanism. The team construct abstracts a group of processes that cooperate in order to realize a subgoal of the system. A team declaration is a named collection of roles, which are either processes or procedures. The body of a procedure is identical to that of a process, but it is a formal entity, which another (active) role from outside the team must invoke in order to begin its execution. A team often represents a layer in a distributed algorithm in which some number of processes suspend other activity, execute an entire protocol described in the team, and then resume other activity. Since there can be numerous instances of the same team declaration, and each procedure in an instance can be repeatedly invoked, the team provides a valuable abstraction for algorithms involving collections of processes so that they can be reused. By defining within a team those new processes relevant to the task of that team, detail unnecessary to the global specification and reasoning is kept internal to the team.

The team construct in itself does not provide a primitive operation for synchronization and communication among roles. Instead it assumes the existence of such an operation that will be used within the roles of teams. Note that the methods of communication or synchronization are not necessarily connected to the unit of abstraction seen in a team. A design decision to form a team to hide information, or to group together certain roles for abstraction purposes, does not have to commit those processes either to a full joint synchronization or communication, or to shared variables. Moreover, if complete access to all variables were assumed within a team, the problem of interprocess communication would be pushed back to the shared-memory model, requiring primitives for mutual exclusion, such as semaphores or monitors. Such a difficulty arises in the guardian construct of Argus [23].
In this paper the basic building block of a team is another construct called the *multiparty interaction*. This construct is used within team declarations to synchronize and communicate among roles, as a primitive atomic operation. Rather than restricting consideration to the common two-party communication, as seen in CSP [21], CCS [24], or Ada, an interaction allows multiple roles within a team to synchronize and exchange values, as one operation within the protocol described by the team.

One of the common refinements to be seen in the section on design methodology is to replace a multiparty interaction by activations of roles in a team containing interactions having a smaller number of participants. This refinement is necessary when the final target system only supports interactions among a small number of participants (e.g., two). By treating details of role coordination through such refinements in late stages of design, multiparty interactions allow simplifying the early design.

The third type of modularization needed is also strongly influenced by the design environment. The subgoals to be satisfied by a system often cannot be divided into layers of execution to be executed separately by the roles of a team while other activities of the invoking processes are suspended. Often, an additional functionality is needed that can best be achieved by adding on to an existing team so that new operations will be interleaved with the old, over extensions of the original states. A classic example achieves clean termination of the roles in a team, without otherwise affecting their previous computations [10, 17]. One way to do this, of course, is to simply implant additional sections of code in the declaration of the roles of the team. However, this obscures the fact that the additional functionality can often be separately designed, and can be reused in many contexts. The *superimposition* construct provides a way to independently declare a module intended to be combined with the roles in a team to create an *extended team*. Although numerous instances of algorithms intended to be used as superimpositions have been developed in recent years, relatively little attention has been paid to language constructs for this purpose.

In the paper all three of these constructs are described in more detail, first the multiparty interaction, then the team, and finally the superimposition. However, it is clear that numerous possibilities exist, and that we have chosen a particular mix of ideas. The fundamental point of the paper is that constructs of each of these types are needed, and that it is not desirable to combine them into one "superconstruct." Thus the joint actions of [2] or [27], and the rendezvous of Ada or its multiway generalization [8], lead to over-specification by not separating the abstractions seen in the team from those in the interaction, and neither treats the modularization achievable by superimposition. In Section 5 the design methodology underlying these constructs is summarized, and correctness-preserving transformations replacing one construct by others are shown.

Variants of the new constructs seen here are found in the Raddle language developed at MCC [14] [1]. However, in order to avoid extraneous details of that language, the sequential parts of the programs not involving the new constructs assume a
Pascal-like syntax within roles, along with guarded commands [9] [21]. Previous work has considered the interaction and the team mainly through examples. Here, the properties of these constructs are examined, some partial correctness proof rules are provided, and the superimposition language construct is introduced. Moreover, the transformations and their correctness criteria demonstrate the integration of these three constructs in program development.

2. Multiparty interactions

2.1 Definition and proof rules for interactions

The first modularization concept we describe is the interaction primitive for synchronization and communication. As noted above, the interaction is a basic building block of more complex distributed algorithms. In order to allow meaningful comparisons and complexity measures based on the number of interactions, it should be implementable by a fixed number of lower-level communications, dependent only on the number of participant roles and number of values needed in the interaction. As a natural generalization of a CSP handshake, an interaction can only take place when all participant roles are at a point in their code where that interaction is a possible choice for the next statement to be executed. Within an interaction, multiple values are transferred among the participants. This is represented as assignments to variables. To encourage modularity in the roles, the assignments that change the local variables of each role are given separately in the body of that role, rather than expressing all the changes in one central location. This allows purely local modification of the role, without affecting other roles in the same interaction. An interaction name may appear at several points with differing assignments within the same role, with the one to be executed dependent on the control location of the role when the interaction is to be executed. This resembles the possibility in Ada of several accept statements for the same entry, or the presence in CSP of several matching communications for the same tag.

The interaction construct embodies an unusual philosophical view of communication as “taking” what is needed rather than “sending” the appropriate values, as seen in common message-passing operations. Participation in an interaction means that the values of the variables of that role are momentarily available to other roles, and that the role will 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 include variables from other participants, only for purposes of reading. Although a role must therefore know some variable names from other roles, this can be restricted to the subset of variables needed for the interaction. 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. A 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.
In the example seen in Figure 1, when the executions of \( P, Q, \) and \( R \) are at a body of the interaction \( \text{inter} \), the role \( P \) will update its local variable \( x \) by using \( y \) from role \( Q \) and \( z \) from role \( R \), while in the same interaction \( Q \) updates \( y \) using \( x \) from \( P \). If the control of \( R \) is at the first appearance of \( \text{inter} \), then \( R \) merely provides values and does not update any of its local variables (and thus corresponds to a role that is sending values in a message passing point of view). On the other hand, if \( R \) is at the second appearance of \( \text{inter} \) when the interaction is chosen for execution, then the other processes act as previously (using the value of \( z \) at this point), but \( R \) copies the value of \( x \) from \( P \) to its local variable \( z \).

The list of participant roles in an interaction can either be inferred syntactically as all roles with a body of that interaction, or, preferably, explicitly listed in an interaction declaration. This second option allows syntactic consistency checks, and also permits restricting the variables made available by each role, to increase security and privacy (see [12]). The interaction provides synchronization capabilities by requiring a complete synchronization of all the participant processes at the beginning of an interaction execution. Interaction bodies can serve as guards in a nondeterministic choice statement, analogous to the appearance of input statements in the guards of CSP or Occam, or the Ada select statement. The choice of which interaction to execute from among the possibilities at a given time in the system is left to the scheduling mechanism of the implementation. Both centralized and distributed schedulers are possible, and this issue is not considered here.

Historically, message passing has been viewed as the natural mode of communication between distributed processes. We claim that this is because the network architectures presently available for implementations of distributed languages encourage such a view. In fact, it is often natural to the problem to consider multiple processes "coming together" to exchange information in an atomic operation, without having to partition the participants into "senders" and "receivers." As will be seen in later sections, a multiparty interaction can serve as an intermediate stage in the development of a program, even if the version to be finally executed is restricted to two-party interactions as

\[
\begin{array}{c|c|c}
\text{P} & \text{Q} & \text{R} \\
\text{var} \ x & \text{var} \ y & \text{var} \ z \\
\text{...} & \text{...} & \text{...} \\
\text{\textit{inter}[ } x:= y+z \text{ \textit{]} } & \text{\textit{inter}[ } y:= y\times x \text{ \textit{]} } & \text{\textit{inter}[ ] } \\
\text{\textit{...} } & \text{\textit{...} } & \text{\textit{...} } \\
& & \text{\textit{inter}[ } z:= x \text{ \textit{]} } \\
\end{array}
\]

Figure 1: A simple interaction
primitive operations. Moreover, with the use of advanced bus architectures becoming widespread, direct implementation of multiparty interactions may be feasible.

A global proof rule for an interaction with a single body in each process is relatively simple. Such an interaction may be viewed as a generalized assignment statement. Therefore, for participants $P_i, 1 \leq i \leq n$, if each participant contains

$\{ p_i \} \text{inter}[x_i := e_i] \{ q_i \}$

in a proof outline [25], then we may simply define the rule

$\{ \wedge_{i=1}^{\alpha} p_i \} x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n \{ \wedge_{i=1}^{\alpha} q_i \}$

for all $i, 1 \leq i \leq n$, $\{ p_i \} \text{inter}[x_i := e_i] \{ q_i \}$

However, the above rule is unnecessarily global and a more modular version can be devised that does not consider all assignments at once. The modular version does take into account the possibility of a process having several appearances of the same interaction name, each with a different body. When that possibility does occur, each interaction body for the same interaction within the same role should have the same precondition, although the postconditions may differ. This is necessary so that the correctness reasoning for role $A$’s part of the interaction is not dependent on the internal decision of role $B$ of which interaction body of $B$ will execute. If there are $n$ participants in an interaction, and each participant $j$ has the same precondition $p_j$ before each body of the interaction in $j$, then for each body $b$ it can be separately shown that the global precondition will lead to the local postcondition. That is, if $I$ is a global invariant proven separately,

for each interaction body $b$, $\{ I \wedge \wedge_{i=1}^{\alpha} p_i \} x_b := e_b \{ q_b \}$

for each participant $j$ with interaction body $b$, $\{ p_j \} \text{inter}[x_b := e_b] \{ q_b \}$

The point of this rule is that since the initial values are the only ones used by other participants in an interaction, only these must satisfy a uniform precondition (the "expectation"). However the result computed in an interaction body within a role $R$ is not globally used, and can be appropriate to its particular context within $R$.

A more complex proof system that integrates interactions into a two-level cooperating proof, with a local proof and a separate cooperation test, may be found in [18]. Some generalizations of interactions are considered in [12].

2.2 Examples

Many tasks that are difficult in distributed systems with a nonblocking send operation are simple to express using interactions. Among the solutions to abstract versions of
these tasks (assuming a nonblocking send) are algorithms to take a global snapshot [4], to choose a leader among processes organized in a ring [26], or to resolve resource conflicts in a fair way [5]. Even a synchronized two-party interaction (as in CSP) is insufficient to remove this complexity, but interactions involving all the participants make these tasks trivial. The usual reason for rejecting this global view is that such operations are either not available, or are too expensive. Here the operation is available, and the possible cost is dealt with at a later stage of the methodology. Thus Figure 2 shows an interaction to take a global snapshot, followed by some unspecified local computations, and then an interaction to broadcast the result of this computation from $P$ to the other roles, for three participants. The generalization to $n$ is obvious: all except one of the participants merely make their local state available, and later record the result, while the final participant records the global state, computes, and makes the result available to the others. It remains only to code the roles so that the interactions are guaranteed to occur when such a snapshot and subsequent broadcast are needed.

Other simple interactions describe additional well-known distributed tasks. For example, global conflict resolution using priorities is achievable by computing a minimal value within an interaction involving all the processes.

There are situations where an interaction is needed involving neither all possible roles, nor, at the other extreme, only a pair of roles. For example, a system for Electronic Funds Transfer (EFT) typically might have many processes for treating the databases involved, auditing the transactions, checking balances and credit, etc., yet have an interaction involving three participants, a Source-bank, a Target-bank, and an Initiator. Within the algorithm that checks the balance of the Initiator in the Source-bank and sets up the transaction, the actual transfer of cash can be seen in the transfer interaction of Figure 3. Note that the Initiator provides only the value of requestval (presumably after various checks have been made) and synchronizes with the other participants. If the total amount of money in the system is to be conserved, it is crucial that the interaction transfer be viewed as an atomic event, and that when it is completed, the

```
P var s, x
... snap[ s := x, y, z ] ...
... broad[ ]
Q var y
... snap[ ] ...
... broad[ y:=s ]
R var z
... snap[ ] ...
... broad[ z:=s ]
```

Figure 2: Interactions for global snapshots and broadcasts
Initiator can depend on the fact that the transfer of funds has actually taken place. Nevertheless, subsequent transformations may partially relax this atomicity, as will be seen in Section 5.

2.3 Implementing interactions

In order to justify the primitive atomic nature of an interaction, a typical implementation is given here for a centralized scheduler. A lower-level distributed language based on message passing with a nonblocking send operation is assumed. In this case, the interaction view of "taking what is needed" must be converted to the more traditional "sending" view. That is, during compilation the potential needs of each participant are first identified for each interaction. In particular, for an interaction inter and process P, the union of the variables in the bodies of inter in P is computed. Then the variables not local to P are divided into subsets according to the source of the variable, i.e., the participant in which the variable is defined. Each subset represents the variables to be transferred from a process R to P in order to execute a body of inter. This subset will be used by the implementation of participant R to define a package of values to be sent to P when interaction inter is chosen for execution.

During execution, whenever either an interaction as a statement or a choice point with interactions as guards is reached, information on the locally possible interactions is sent to the scheduler process, and the implementation of the participant waits for a decision. The scheduler decides which interaction is to be executed, and sends messages to that effect to the implementations of all participants. These then prepare the various packages of their local values needed by some of the other participants, and send each package as one or more messages. When an implementation of a participant receives the
values it needs from the others, it executes the local assignments, and immediately continues executing the code after the interaction.

Clearly various code improvement techniques may be applied in order to avoid transferring large amounts of data unnecessarily when the variables are data structures. For example, the implementation of a participant can record whether there have been changes in a variable needed by another participant since the value was last sent, and if not, send only an indicator to use the previous version.

Note that although there is no synchronization at the end of the implementation, an interaction may still be viewed as an atomic event. This is because the separate continuation is nonobservable. It cannot be distinguished from the case in which all the participants finish synchronously, but wait at that point arbitrarily long before separately continuing.

3. Teams

3.1 A description and proof rule for teams

As was briefly described in the Introduction, a team declaration in Raddle is composed of a collection of roles, which are either processes or procedures. Each role has a separate state space and executes asynchronously. All communication and synchronization among roles in a team is done through interactions, where the state is temporarily extended as described in the previous section.

An instance of a team can be created dynamically, and at the moment of creation the processes of the instance are activated and begin executing their code. Typically, a variable will be declared at block entrance to be an instance of a team, and it cannot be changed within the block. A copy of each procedure becomes available for enrollment, or activation (which are used interchangeably in the continuation). Any active role from outside the instance of a team can enroll in a procedure in the team instance, by writing \texttt{instancename.procname(actualparameters)}.

Upon enrollment, parameters are passed by value and the procedure immediately begins executing, without delaying until all of the procedures of the team instance have been activated. However, when a process or procedure reaches a point at which it may continue only by executing an interaction with participant roles (procedures) not yet activated, it must wait. This convention is in fact a natural consequence of the fact that an interaction may be executed only when the controls of all participant roles are at a body of that interaction.

When a role $R$ in team $A$ activates a procedure $P$ of team $B$, $R$ suspends its execution until the procedure $P$ terminates that activation, as described below, and then continues executing. When a return statement is reached, the role has completed executing for that activation. A process then will simply terminate, while a procedure will first pass any result parameters back to the external process or active procedure that activated it.
Of course, a terminated process cannot participate in any later interaction, so care must be taken to avoid deadlock situations.

Nothing stated so far prevents having several activations executing the same procedure at the same time. Each such activation has its own set of local variables and control, in an activation record. However, the various activations might 'compete' over which will participate in an interaction with other roles. For complex teams, this can make it difficult to guarantee that there will be a sequence of interactions with the same activation, even when that is required. When only one activation at a time is desired for procedures, this can be enforced by a superimposition adding interactions which simulate a binary semaphore to prevent multiple activation. In the continuation, we assume single activations. Proof rules and transformations that do not have such a restriction are under development.

Typical team declarations are shown in the following subsection. They contain both internal processes and procedures callable from outside. Within the code are interactions and enrollment commands to procedures of other teams. Aside from the header, there is no difference between the statements allowed in a procedure and those seen in a process.

A team with no procedures will naturally serve as the top level in a system design. That is, such a team has only processes, and these will presumably activate the procedures in other teams. At the other extreme, a team with only procedures closely resembles the script concept [19], generalizing the usual notion of procedure to the distributed context. Such a team can be considered to be passive, awaiting activation after its declaration. In the more general mixed case, new processes are introduced only at the level they are needed, and are hidden from consideration outside the team that contains them. Thus in the classic buffer example, with an internal buffer process, teams outside do not need to be aware of the existence of a buffer process, only of the procedures Producer and Consumer.

A partial correctness proof rule for teams can be complex, but is similar in spirit to those for procedures with parameters [20]. Some simplifications are possible because parameter passing is by value-result, and there are no global variables or aliasing. Here we will give a simple version appropriate for a team in which the roles are activated once, and all of them terminate with a result satisfying an output specification (a postcondition). The activations of a team are in rounds if each procedure is activated exactly once between the time that a first activation occurs and the time that the last procedure terminates for that round. This sort of activation sometimes arises when teams are substituted for interactions. For a team \( T \), if \( P_i \) for \( 1 \leq i \leq n \) are processes with bodies \( PS_i \), while \( R_i(u) \) for \( 1 \leq i \leq m \) are procedures with formal parameters \( u \), bodies \( RS_i(u) \), with preconditions \( pr_i(u) \) and postcondition \( qr_i(u) \), then let \( I \) represent an assertion involving only the variables of the processes. We have
The invariant involves only variables of the processes because the procedures have a new activation record for each activation, and there is no concept of an "own" variable within a procedure. Processes, on the other hand, are ongoing, and the possible states they may be in during procedure initiation and execution are expressed using \( I \). The invariant is established by showing that it is true of the processes when a team instance is created, and that each operation of a process maintains it. Note that the preconditions of the procedures may be used in establishing the invariant.

Recall that the conditions for applying this proof rule are very restrictive. Often several activations of one role are needed before another role can terminate with its postcondition true, and this possibility is not covered by the rule. Thus, the rule above is sound but incomplete. Extensions to treat the more general case have the same basic structure. Note also that at least the processes of a team are likely to be nonterminating and reactive in nature. That is, the processes often serve the procedures by participating in interactions with them, then continuing to execute some internal computation, and waiting at one or more interaction bodies with procedures as participants until those procedures are again activated. In addition to the safety properties, significant effort is needed to prove that the team procedures will indeed terminate and that each needed procedure will be activated, as will be seen in Section 5.

```plaintext
team broadcast;

procedure source( val: integer; n: integer );
var index: integer;
begin
  for index := 1 to n do
    send[ ]
  end;

procedure target( recval: integer );
begin
  send[ recval := val ]
  end;
```

Figure 4: A team for simulating broadcast to \( n \) targets
conditions for top-down design using these constructs. Related ideas of splitting into processes or of distributing previously centralized actions may be seen in [6] and [2].

5.2 Splitting interactions

During early design stages, some participants may be included in the same interaction even though they do not require direct communication or synchronization. This may be done in order to simplify the initial design and correctness reasoning. We first deal with the simplest such case: A and B are a pure split of an interaction I iff A and B partition the participants of I (i.e., are disjoint sets of participants of I with every participant in one set) and the following two conditions hold:

1. Each participant of I uses only variables from the set of participants to which it belongs, and
2. Synchronization between the roles in A and those in B is not essential. This safety criterion intuitively means that the team would still be correct with respect to its specification if the interaction bodies were not synchronized. It is further defined below.

If A and B are a pure split of I and the interaction bodies do not appear in guards, then the interaction name I can be replaced by IA within roles of A, and by IB within roles of B.

When the synchronization is essential, this transformation is not justified. For example, consider a specification requiring that a variable count in an audit role be exactly equal to the number of updates of a variable from another role, X. The obvious way to achieve this is to increment count and update the variable within the same interaction (with bodies in audit and X, respectively). In this case, the above split of this interaction is not possible, even though audit and X do not use each others variables. The problem is that when one of the resultant interactions has been executed but not the other, there is a state where the value of count will not precisely reflect the number of updates of the variable in X, violating the specification.

A generalized split generalizes a pure split to n interactions, with sets of participants no longer necessarily disjoint. A few preliminary definitions are needed. An assignment statement in an interaction is associated with a set of roles if all variables in the assignment are from roles in the set. For each interaction, a role belongs to exactly the sets with which the assignment statements of its bodies for that interaction are associated.

A synchronization among roles \( r_1, \ldots, r_m \) (a subset of participants) is essential with respect to a specification if an invariant needed to prove the specification is no longer invariantly true when the synchronization is not made, even though the roles have the same local sequences of values. This is a reformulation of the condition (2) above. A set of interactions is linked if whenever one of the set is executed, then all of them will be, and moreover in each such round no additional communication can occur (even
indirectly) between participant roles that have executed an interaction of the set, and those that have not yet. This definition is used to show when a generalized split is justified. Note that even when a set is linked, both local actions and some “irrelevant” interactions not in the set can be interleaved with the execution of the interactions in the set.

If all assignments in all bodies of the interaction can be associated with a set, and all essential synchronizations are among roles in the same set, a split using those sets can be made. If the new interactions to be described below are linked, then the transformation preserves correctness. This is further discussed after the transformation is described.

For roles in only one set, the treatment will be as for a pure split. All its assignments must be associated with the set it is in, since the left-hand-side variables in assignments are clearly local to the role. The name of the interaction will simply be changed to that of the new interaction associated with the appropriate set. For those roles in the intersection of the sets, the transformation is more complex. Those roles will have to participate in a new interaction for each set the role is in. This requires duplicating some variables, and building a small loop that allows choosing the interactions once each, in any order, for those in the intersection.

In particular, each participant is transformed according to the following steps:
1. For each interaction body, divide the assignments among the new interactions so that each is in the interaction corresponding to the set it is associated with.
2. For each variable \( v \) on the left-hand-side of an assignment in the original interaction bodies, define an auxiliary variable \( v' \), and substitute \( v' \) for the left-hand-side appearances of \( v \) in those assignments.
3. Build a nondeterministic loop to execute each new interaction body once, under the same local conditions as those for selecting the original interaction body.
4. After the code of the above loop, add assignments from the auxiliary variables \( (v') \) to the original variables \( (v) \).

An example of an interaction among four participants is shown in Figure 8. If, for example, only the synchronization between \( W \) and \( Z \) is essential to the correctness, a split can be made to \( A = \{W, X, Y\} \) and \( B = \{W, Z\} \). The transformation then leads to the version shown in Figure 9. In this simple case, rather than building a loop, it may be more convenient to list the two possible sequential orders of choosing the interactions. Note that the transformation has the effect of allowing role \( Z \) to proceed after its interaction with \( W \), even if \( X \) and \( Y \) are not yet ready to execute their interaction with \( W \), and vice versa.

The following claim is now true:

Claim: Under the conditions for splitting, if the resultant interactions are linked, then each new computation sequence differs from one in the program before the transformation only by reordering of independent operations from different roles. In this case, for
each combination of interaction bodies, instead of the original interaction all of the new interactions are executed in arbitrary order.

Proof: The claim follows from the observation that the requirement of linked interactions is a generalization of the concept of a communication-closed layer [11] to the context of multiparty interactions. This means that we can always rearrange independent operations to give a computation in which all of the linked interactions occur without interleaving of additional operations. It also follows that each individual role will have the same possible sequences of values as before the transformation.

One (extremely restrictive) way to guarantee linked interactions after the transformation is to forbid any explicit nondeterministic choice at all in the team before the transformation. In that case, if one role reaches the point where the original interaction occurs, then all must reach one of the interaction bodies, or deadlock will occur. After the
transformation, this remains true, ensuring linking, as long as deadlock freedom held for the original team.

The transformation above assumes that an assignment statement from the original interaction will be executed unchanged in one of the resultant interactions. If a finer grain of breaking synchrony is desired, such as only having two-way interactions, this can be achieved by first manipulating the local assignments. That is, the assignment \( w := x + y \) could be replaced locally within \( W \) by the copy operations \( wx := x \) and \( wy := y \) inside the interaction, plus the local assignment \( w := wx + wy \) after the interaction. Then applying the generalized splitting transformation could result in an interaction between \( W \) and \( X \) and another between \( W \) and \( Y \).

There are also situations in which it is not necessary for each of the new interactions to execute the same number of times. In this case, the transformation is simpler for those in the intersection of the sets (it is often merely a nondeterministic choice among the interactions of the sets it is in). Linking is also not required for correctness. Of course, this looser transformation can lead to global states and values for variables that were previously impossible.

One example of a class of programs for which this greater relaxation is appropriate is for control based on sampling of sensors. In an initial design, all sensor processes can provide values simultaneously to an evaluator, within an interaction. This means that the values provided are dictated by the rate of the slowest sensor. When this is split into binary interactions with the evaluator, each sensor can provide updates as they become available. The relative numbers of such updates for each sensor is irrelevant, as long as each sensor is able to interact reasonably soon after a new value is available. Other examples relate to relaxations of synchrony for matrix iteration problems [14].

Of course, even more complex substitutions are possible, including a universal transformation based on the ideas in Section 2.3 on the implementation of interactions (where a multiparty interaction is simulated by message passing). However, such general protocols are best left to a simulation team, which is used to replace the interaction using the transformation in the following subsection.

5.3 Substituting procedure activations for interactions

The same basic ideas seen in the generalized splitting of interactions can be used to substitute role activations for interaction bodies. The goal is again to relax synchrony, and to replace an interaction by an entire communication protocol. The transformation can exploit the modularity and reusability of teams, as well as the option of introducing new processes within the team.

Some difficulties that were taken care of automatically by the nature of the specific transformations used previously must now be treated explicitly. For example, since interactions always terminate, there was no proof obligation relating to the termination of the transformed team. For role invocations this must be shown explicitly. Similarly, since
we used the same assignment statements, the safety properties were automatically main-
tained. This will no longer be so. The following conditions are natural consequences of
such considerations.

Activations of procedures in a team $T$ are fully substitutable for the bodies of an
interaction $I$ appearing in a team $C$ if the following three conditions are satisfied:
safety—All pre- and postconditions of interaction bodies of $I$ needed to prove properties
in the specification of $C$ can be proven for the appropriate procedure activation. All glo-
bal invariants needed to satisfy the specification are also true throughout the procedure
execution.

internal termination—If a procedure of $T$ is activated for each participant in the interac-
tion $I$ in a way consistent with the original interaction bodies, that activation of a pro-
cedure will terminate. Note that this guarantees both deadlock freedom and the absence
of infinite computations for those procedure activations.

linking—In the context $C$, if any former participant of $I$ executes a substituted procedure
activation, all former participants will do so, under the same local conditions as the
interaction bodies, and without communication among those that have begun a substi-
tuted procedure, and those that have not.

If linking did not hold, as previously, some of the participants might activate the
procedures, while others choose different alternatives. This clearly could lead either to
deadlock or at least to a nonequivalent result. Note that linking is a property of the con-
text of activation, and cannot be checked by examining the interaction bodies or the team
with the procedures. Note also that if the interaction does not require a full synchroniza-
tion of all its processes, the previous transformations can be used to split it into interac-
tions of a lower arity. In order to separate concerns as much as possible, this is not done
within this transformation.

Linking becomes crucial when interactions appear in guards. The need for substitut-
bility indicates that procedure activation should also be allowed in this context, but
then extreme care must be taken to avoid deadlock situations. Another source for such
problems is when the same role of $C$ participates in an interaction several times. In the
original team, no confusion could result. However after the substitution of procedure
activations to $T$, the same role of $C$ could successively activate roles of $T$ as if it were
different participants of the original interaction.

For example, if we wish to replace a broad broadcast interaction among $n$ partici-
pants (as seen in Figure 2) by the team with two roles in Figure 4, the activation
broadsim.source($s,n$) appears instead of the body broad $I$, while activations of the role
broadsim.target(localvar) replace the bodies broad $I$ localvar := $s I$. Note that although
both safety and internal termination are easy to show, it could happen that only one pro-
cess receives the value of $s$ ($n$ times), if linking does not hold.

As previously, there are contexts in which it is not necessary to guarantee that all
assignments of the original interaction are executed even when some are. Again, in such
6. Conclusions

One of the main problems with existing design methodologies is that the entire specification must be made at the highest level in order to guarantee that subsequent refinements are correct. This is unrealistic and is one reason that formal methods are rarely used. The constructs suggested here allow postponing decisions and later substituting or adding components. The separation of concerns and the modularity of introducing new processes, procedures, and data structures only on the needed levels encourage practical stepwise development in the difficult distributed context.

Moreover, the correctness criteria seen above help to show the range of applicability of various transformations, relative to a specification. Usually, full substitutability of a team for an interaction is unnecessary. Sometimes forms that weaken synchrony are still correct in particular contexts (i.e., relative to a particular specification), and allow greater parallelism in execution. Similar considerations apply to potential effects of a superimposition on the specification of a basic team, or possible splittings of multiparty interactions.

References


3.2 Examples

Teams are commonly used to capture patterns of communication among processes, to allow reuse. Thus if we wish to express a pattern transferring a value from a source to a collection of targets, using only two-party interactions, the team seen in Figure 4 could be used. Here only procedures appear in the team: one is a source role to provide the value val and the other is a target role to be filled in turn by the recipients of the value. Any role external to the team may enroll either as a source or as a target. Once a source has been activated, the activating role can return to continue executing its code only after there have been n enrollments in the target procedure. This constitutes a simulation of a broadcast operation from the role that enrolls as the source, to the collection of roles enrolling as a target. Note that if some role activates the target role several times, the team will not simulate a broadcast to n different targets. This is considered in Section 5.

Teams are also commonly used to implement data structures that serve roles from other teams. The team stackimp, seen in Figure 5, illustrates a team with both procedures and processes. Roles external to the team activate the procedure push to push a value to the stack, and the procedure pop to receive a value and pop the stack. The internal organization, or even the existence of an internal process stack, is of no interest to users of the procedure roles. The internal process is created along with an instance of the team, and it does not terminate. In the particular team given here, the user of the pop procedure will either obtain an element (if the stack is nonempty and the remove interaction occurs) or receive an announcement succ = false (if the stack is empty and the check interaction occurs).

Finally, a team encapsulating the electronic funds transfer example seen in Section 2 has the procedures sourcebank, targetbank, and initiator. For example, sourcebank might obtain the account number and amount to be transferred from the initiator, check the balance and/or credit line of the account by enrolling in a role creditcheck of another team, and then either force executing the interaction cancel with the other two processes, or force executing the transfer interaction seen in Figure 3. The initiator would participate in interactions to make available the needed information to the other roles, and then be ready to participate either in a cancel interaction or in a transfer depending on the decisions of the other roles. A process representing an actual bank could then enroll as a sourcebank at one point in its execution, and later, after terminating that role, enroll as a clientbank.

The team might also include an additional internal process called audit that acts as a "silent participant" in all transfer interactions, by recording in a local variable all account numbers and amounts transferred. This process may compute statistical summaries and activate an external role makereport from time to time in order to print out its information. Again, the monitoring done by the process audit is not externally visible to the users of the procedures.
team stackimp(n: integer)
process stack;
var a: array[1..n] of integer;
c: integer := 0;
begin
repeat
  [c<=n & insert[ a[c]:=val ] -> c:=c+1]
  [c> 0 & remove[] -> c:=c-1]
  [c = 0 & check[] -> skip]
end;
procedure push(val: integer);
begin
  insert[]
end;
procedure pop(v: integer; succ: boolean);
begin
  [check[] -> succ:= false]
end;

Figure 5: A team for implementing a stack

4. Superimpositions

4.1 Motivation and description of superimposition

Neither multiparty interactions nor teams are appropriate for expressing a type of modularization which has become common in distributed programming over the last decade. In this category are separately designed algorithms intended to be "added on" to other basic algorithms, to create a combination satisfying additional properties. We suggest a mechanism for separately describing the needed additions, generically showing the correctness, and reusing the module in a variety of contexts. The construct is called a superimposition, and it consists of both a declaration and a binding mechanism between teams and superimpositions. The result of such an activation is known as an augmented team.

An early use of the term "superimpose" is in the termination detection algorithm of Dijkstra and Scholten [10]. Many other examples are explicitly intended to be used as
superimpositions, starting with the termination detection algorithm of Francez [17] for nondiffusing computations, algorithms for deadlock detection ([17] and many others) and the global snapshot of [4]. Superimposition is also appropriate for describing more mundane tasks such as monitoring, accounting, and debugging other algorithms. These and many other examples are characterized by the separate description of the algorithm to be superimposed, and a rather informal description of the way in which the algorithm is to interact with the other tasks being performed in the processes. The sometimes implicit assumption is also made that the result of superimposing is a complete interleaving of the basic computation and the superimposed code.

The notion of superimposition (but termed superposition) has been investigated by Chandy and Misra in the context of their UNITY approach to programming [5] [6]. An abstract view of superimposition with somewhat different properties than the version seen here, and with no parametrization, has been developed by Bouge and Francez [3]. A language independent view of a construct for superimposition may be seen in [22].

A superimposition declaration describes a distributed algorithm intended to be combined with the roles in a team. The declaration is composed of role (both process and procedure) declarations and a collection of roletype declarations. The role declarations differ from those in teams in that additional binding parameters are used so that identifiers, including interaction names, can be bound to identifiers in the basic team. Rotetypes differ from roles in that they are intended to augment a regular procedure or process from a basic team, and this is reflected in the syntax and semantics.

Rather than directly instantiating a superimposition declaration, a combination operation must occur between a superimposition declaration and a team declaration. The result of such a combination is a new team declaration, which can have several instances in the usual way. When such a combination is declared, part of the state of each role of the original team is associated with formal variables in one of the roletypes of the superimposition declaration. Part of the code of the roletype at least conceptually modifies the code of the original role, while other parts are executed in an instance of the resultant team, as if they were additional top-level options to the original role code. The new roles added by the superimposition have their formal parameters instantiated to variables of the basic team and in particular to interaction names, so that joint interactions can be defined.

When the declaration is combined with a specific team, the binding of the parameters to variables in the roles of the basic team is also described. In order to reason about a superimposition independently from a specific basic team, the requirements of the superimposition from any basic team with which it may be combined must be specified, and the adaptation must be shown to have the desired effect for a wide family of basic teams.

A syntax for a roletype within a superimposition declaration is seen in Figure 6. It is easiest to understand the semantics of a superimposition declaration by giving an
superimp \( <\text{superimpose}> \langle <\text{parameter list}> \rangle \)

\text{type-declarations}

\( \text{roletype} <\text{name}> ( <\text{parameter list}> ) \) \{, \langle <\text{restrictions}> \rangle \}

\text{var} \langle <\text{local declarations}> \rangle

\text{transform}

\{ \langle \text{description of transformations to the basic role} \rangle \}

\text{initialize}

\{ \langle \text{code to be executed before the basic role begins} \rangle \}

\text{add}

\{ \langle \text{new code for this roletype, to be interleaved with basic role} \rangle \}

\text{finalize}

\{ \langle \text{code to be executed after the basic role is completed} \rangle \}

\text{endrole}

\text{roletype} ...

\text{process} ...

\text{endsuperimp}

\begin{figure}[h]

\centering

\includegraphics[width=\textwidth]{superimpose.png}

\caption{The form of a roletype in a superimposition declaration}
\end{figure}

The new team \( \text{New} \) has all roles of \( \text{Super} \) after substituting the parameters from \( \text{Team} \), plus all augmented roles of \( \text{Team} \).

To obtain the augmented roles, each \text{role} of \( \text{Team} \) is transformed as indicated in the \text{transform} section of the associated roletype (after substituting parameters). Usually this involves gathering additional information not needed by the basic team, but essential to the superimposition, or occasionally stopping the operation of a role from the basic team, so that the code to be superimposed can be executed. This has the general form \text{label: code} \rightarrow \text{newcode}, indicating that the part of the basic role matching the left side will be replaced by the code on the right side. The allowable transformations are generally restricted in order to prevent interference with the values being computed in the basic role. In addition, the code in the \text{initialize} section is added so that it will execute...
before the code of the original role. The code of the add section will be additional alternatives to the top level of the role (which usually has the form of a collection of non-deterministic alternatives in a reactive system). Any return statements in the role of Team will be preceded by the code in the finalize section of the corresponding roletype.

As long as the superimposition does not change variables of the basic team, and only uses them, or adds new variables and statements for its own purposes, the continued partial correctness of the basic team is guaranteed to hold also in the augmented team after the superimposition. The correctness of the superimposition with respect to its added functionality may be shown "generically" using assumptions about the nature of any basic team upon which it may be superimposed.

4.2 Implementing superimposition

The obvious view of superimposition is as a macro facility that builds a new team by transforming the basic team as indicated above, adding the new roles, and adding the code indicated by the appropriate roletype to each basic role.

From the definitions, it is indeed required that the combination will execute as if this new team were actually generated, but in fact the code from the superimposition need not be implanted in the basic team. For reasons of efficiency, both in set-up and in space occupied by the resultant code, other approaches may be preferable. Instead of implanting code with actual parameters substituted, the implementation might generate internal transfers of control to procedures of the implementation of the superimposition. These resemble the "thunks" used to implement a call-by-name parameter passing mechanism. In other words, the implementation of the augmented role would include special procedure calls that are activated from time to time. Such a procedure transfers the parameters needed, executes one of the commands in the add section, and then returns the updated values to the implementing role.

4.3 Examples of superimpositions

A common application of superimposition is to express additions needed to monitor various activities of teams. This could be for purposes of debugging, accounting, statistical analysis, or auditing. Thus in the banking example seen previously, it is reasonable to define an audit superimposition which can be added on to teams conducting various forms of transactions. As a simple example, let us assume that it is desired to record the value of some key variable, whenever some crucial interaction occurs, where both the variable to be recorded and the interaction are to be given as parameters. In this case the superimposition seen in Figure 7 might be used. Here a statistics summary is prepared every 100 interactions, and the roles of the basic team are assumed to eventually terminate. Note that when count = 100 the boolean part of the guard prevents additional inter interactions from executing until the report is sent in the second alternative guard.
5. A transformational development methodology

5.1 Overview

This superimposition would be applied to the banking team by writing

```
auditbank is bank using simaudit(requestval, transfer)
```

Where `Target, Source, Initiator` includes `participant`.

In this trivial example, the participant roletypes are unaffected, except to add the interaction `done` before terminating. This will make all of the participants synchronize at their termination points, in addition to informing the new process `gather` that it should make one last report and itself terminate. Of course, a more complex superimposition could use two-party interactions to pass this information without requiring full synchronization. Also, if information were required that is not in one of the variables of the basic team, such as the control location of the participant, this could be incorporated. The participant roletype would then include a `transform` section to add the variable `location` to each role, and increment it at crucial labels. This variable would then also be recorded in the interactions. If the values of local variables were to be recorded not as part of existing interactions, then new ones could be added especially for this purpose, in `add` statements of the roletype. In complex superimpositions such as those for increasing fault tolerance or termination detection, the `add` section fills a more central role, since each roletype plays a more active part.
In order to exploit the three constructs described above, we propose a methodology for developing distributed systems by adding on constructs and substituting one construct for another. The idea is to first develop a correct but possibly inefficient design for the most essential features of the system specification, making use of massive information transfers and synchronization. This version may also be incomplete. That is, the design will consist of a team with multiparty interactions containing assignments using abstract functions on the right-hand-side, local actions in the roles, and calls to procedures of teams not yet written (beyond their headers). The incomplete interactions, functions, and teams each have specifications they are assumed to satisfy. This design will be relatively easy to understand and reason about, for the part of the overall system specification it satisfies. The roles correspond initially to abstract entities of the problem statement, which are intended to participate in various activities.

Assuming a full implementation of a language with these constructs, such a design may be executed as a prototype version of the system, to aid in debugging the system requirements. In this case the undefined roles will have superimposed on their (empty) bodies an interface to the user, who during execution will provide appropriate responses to prompts describing needed result parameters for the stubs of activated procedures.

In subsequent development, the incomplete teams are written and additional functionality is added by superimposing on existing teams. In addition, transformations are applied to reduce the information transfer and weaken synchronization when possible. In this framework, some of the multiparty interactions are replaced by several interactions, each with a smaller number of participants. Others are replaced by invocations of roles in teams that simulate the interaction in a sense examined below. Note that as the teams and superimpositions are written, new (internal) processes and procedures are defined, and the need for additional teams arises, so that numerous iterations may be needed.

In order to encourage reusability, libraries of teams and superimpositions may be used. Each entry in such a library includes a (nonexecutable) specification, (e.g., in temporal logic) in order to aid in identifying when it is applicable. Typical entries for teams include distributed data structure implementations (such as the stack example in Figure 6), abstractions of repeated activities such as the banking transaction, and simulations of patterns of communication using interactions with a small number of participants. For superimpositions, various debuggers, statistics gatherers, deadlock detection/prevention algorithms, schedulers, and resource allocators will be in the library. Individual interactions will generally not be sufficiently complex to justify their inclusion in a library. Exceptions may be domain dependent transfers of information that recur often in a system.

In order to define when such a development is correct, it is necessary to understand when an interaction may be split into several smaller interactions and when activations of procedures in a team may be substituted for the bodies of an interaction. In conjunction with the criteria for applying a superimposition to a team, these constitute correctness