Implementing a Language for coordinated distributed programming on a transputer multiprocessor

Allon Adir  Nissim Francez  Shmuel Katz
Department of Computer Science
The Technion, Haifa, Israel

Abstract

*IP (Interacting Processes)* is a high-level language for distributed programming. The unique features of IP are its use of *Multiparty-Interactions* as a communication and synchronization primitive, and the organization of the processes and Multiparty-Interactions in *Teams* that have abstractions of processes known as *roles*.

We describe the issues that have to be faced when implementing such a language on a transputer system based on message-passing, and the solutions seen in an implementation of a version of IP. A version of a scheduling algorithm is presented to select enabled interactions for execution. A new type of analysis known as *conflict propagation* is used to detect and avoid some potential deadlocks that are not trivially obvious from the code. We also describe the translation of assignments in multiparty interactions to messages that transfer data values.

1 Introduction

Two of the main characteristics of the transputer, which play a major role in transputer-based multiprocessor systems, are: *Local memory*, reflected as the absence of *shared memory* in the system, and *Message-passing communications*, reflected in the system as a means of process synchronization and point-to-point interprocess data communication. Transputer systems are usually programmed either by adding message-passing system calls to C or Fortran programs, or by using a language with explicit (synchronous) message passing, in particular Occam[11], which originally was seen as the standard mode of expression for transputers.

Later experience in distributed programming, and in the design of high-level programming languages for expressing it, have shown that the above features are too low-level. Stronger abstractions are desirable, relieving the programmer from much detail that can be handled by a compiler.

In this paper, we describe an implementation on a transputer-based multiprocessor system of a high-level language designed especially for the provision of some higher-level abstractions, enhancing the ease of expressing coordinated programming while preserving the distributed nature of the resulting software system. The language is called *IP (Interacting Processes)* [6, 10, 7]. A comprehensive description of the language, its accompanying program design method, and many related issues, will be presented in [9]. The language can be viewed in a dual way: either directly as a programming language, for expressing final programs (as is done here), or as a language for *executable specifications*, where its designs are refined to lower level implementation primitives. Such refinements are based on synchrony-loosening transformations [8].

The main features of the language, described in more detail in the next section, are the following:
**Multiparty interaction:** This is a natural generalization of synchronous message-passing (which is restricted to two participants) to a construct allowing for synchronization and (symmetric) interprocess communication of *an arbitrary collection of participating processes*. In such interactions, the participant processes all synchronize, each can use the local values of the other participants at the synchronization point to compute local values, and then each process continues locally. By means of this construct, many details of intricate protocols are hidden. Indeed, the coordination problem has been extensively studied in distributed algorithms. The results of these investigations need not necessarily be known to the application programmer, but may be exploited by a state-of-the-art compiler.

**Teams and Roles:** These constructs provide for the *abstractions of subsystems*, allowing for a *single definition – multiple use* of “pieces” of system behavior involving a collection of processes. This generalizes the more traditional process or procedure abstraction, that hides only one thread of control. The need for such abstraction was recognized, for example, by Lamport [15].

The *IP* language also includes a novel modularization construct known as *superimposition*, but since this feature is not implemented in the current system we do not include its description here.

How does the *IP* language relate to programming transputer-based multiprocessor systems? How does it compare to using, for example, Occam?

While *IP* is not specifically "tailored" for transputer-based systems, it is compatible to a large degree with the architectural characteristics of such systems, as the current project manifests. Its major aim is facilitating the systematic design of general-purpose reactive distributed systems. Still, it matches perfectly the locality of memory storage typical of transputer systems, while abstracting from the strict binary handshake interprocess communication with a fixed, small fan-out of the interconnection network.

As research in distributed algorithms show, many of its central problems, such as symmetry breaking and coordination, are rather intricate, with very elaborate, architecture-sensitive, solutions. Such issues are better left to an optimizing compiler, designed by experts, while the end programmer, relieved of this burden, can use the resulting high-level constructs and concentrate on the real problems originating from the application itself.

Overall, the situation is similar to the development of high-level language for traditional computer systems. These languages, while not tailored to a specific register architecture, for example, are compatible with the characteristics of a Von-Neumann machine, allowing its judicious use by providing suitable abstractions. We hope that *IP* will have a similar effect on distributed programming.

Below, after a brief but somewhat detailed description of the *IP* language, we focus on the main issues of the actual implementation on a transputer-based system. The programs used to demonstrate *IP* features are designed to show a variety of interesting behaviors, but are not intended as examples of well-structured *IP* applications, which can be found in [6, 8, 9].

## 2 An Introduction to IP

### 2.1 The core language

We give an informal presentation of the core of *IP* by relating to a specific example program in Figure 1 below.
In general, a core program is composed of a concurrent composition (‘||’) of a number of disjoint \(^1\) processes. The program of Figure 1 includes three processes, named P1, P2 and P3. Each process includes a declaration of number of local variables\(^2\) (e.g., \(x_1\), \(y\), and \(z\) in P1). A process may assign values to its local variables.

At any point, a process may participate in a multiparty interaction. It has to specify the name of the interaction, and its \((local)\) part, which specifies assignments to local variables in terms of variables local to other participating processes. In the example program, P1, P2, and P3 can each participate in an interaction named ‘\(\text{cycle}\)’. Thus, \(\text{cycle}\) is a 3-party interaction with P1, P2, and P3 as participants. When a process reaches a point in its code where a local body of an interaction could be the next action, the process is said to be ready for that interaction. The collection of interactions readied by a process is its readiness set. Because of possible nondeterministic choices, there can be several such interactions for each process. Once readied by all its participants, the interaction is enabled and may be executed. Thus, IP processes synchronize by waiting for enabledness of an interaction.

During the execution of \(\text{cycle}\) in the example, P1 assigns to its local variables \(x_1\) and \(y\) the values of expressions referring to the external variables (with respect to P1) \(x_2\) and \(x_3\). When an interaction is executed, the values of all the variables in the assigned expressions are the values that these variables had immediately before the participants started executing the assignments. Thus, in the example, P1 assigns to \(x_1\) the value 3 (the value of \(x_3\) before P3 assigned it a new value), and to \(y\) the value 6 (the sum of the original values of \(x_1\), \(x_2\), and \(x_3\): 1, 2, and 3). By the time all the participants are done with their respective assignments, the values of \(x_1\), \(x_2\), and \(x_3\) have made a cyclic right-shift and each \(y\) contains their sum. After some further local computation, the processes P1, P2, and P3 “meet” again by participating in another instance of the \(\text{cycle}\) interaction. This time the \(x\)’s make a cyclic left-shift. Multiparty interactions thus embody a view of taking what you need from other participants, as opposed to the sending a message view of the transputer system.

### 2.1.1 Nondeterministic selection

In general, an interaction part may appear also in a guard position. At such a point, the interaction is readied too, and if executed the process “passes the arrow” and continues. This provides a means of continuation selection based on “global agreement”: an interaction guard is passed only if all participants pass their respective similar choices. In the example

\(^1\)That is, without shared variables
\(^2\)Assumed all to be integer valued, for simplicity
in Figure 2, the processes P1 and P2 both include an nondeterministic-selection statement (an NS statement for short) with three guarded-commands separated by '||'. The first guarded command of P1 is \( R := R + 1 \), and it is guarded by the interaction-guard \([R < 3 \& \text{right}[] \rightarrow R := R + 1; L := L + 1; R + L = 0 \rightarrow \text{skip}]\), which means that the command is enabled only if \( R < 3 \) and all the other participants of the interaction right (in this case only process P2) are also ready to participate in it. The NS statement of P1 also includes another interaction-guard and a simple-guard \((R + L = 0)\).

When P1 reaches its NS statement, its readiness-set includes the interaction right (because \( R < 3 \)), the interaction left (because \( L < 3 \)) and the action skip (because \( R + L = 0 \)). If P2 has not yet reached its NS statement, then the simple guard of P1 is the only enabled guard that can be selected for execution. If, on the other hand, P2 did reach its NS statement, then P2 is also ready to participate in the right and left interactions, and therefore all the elements in the readiness-sets of P1 and P2 match enabled guards and there are 3 possible selections that may be made:

1. P1 and P2 choose their first guard. Both participate in the interaction right and then continue into their respective guarded commands.

2. P1 and P2 choose their second guard. Both participate in the interaction left and then continue into their respective guarded commands.

3. P1 and P2 choose their third guard and continue into their respective guarded command (which includes the "no-operation" statement of IP - skip).

The selection of one out of these three alternatives is done nondeterministically. The selection must be done at run-time because the guards may refer to input values, and it must be made in cooperation with all the participants.

Three run-time implementation problems connected with interactions can be observed in this example:

- Each process should be able to determine its own readiness set.
- A selection must be made by considering the readiness-sets of the program processes.
- Once the participants of an interaction have been synchronized, the external values that are needed by each participant must be communicated.
team user(int n, des &a(), des sw)
{
    if(n > 1) {
        a(n-1):=new user(n-1,a,sw)
        n = n-1;
        skip
    }
}

program prog
// Everything to the right of
// two '/' characters is
// a comment.

// a is an array of
// 10 designators
des a(10), sw;
sw:=new switch_team();
a(10):=new user(10,a,sw);

Figure 3: An example of IP teams and roles

IP also has an nondeterministic-loop statement - NL for short (which has the same syntax as an NS statement, except that '*' is prefixed to the statement). The meaning of such a statement is that the selection and execution of the guarded commands is repeated until the readiness set is empty (an empty readiness set in an NS statement is illegal).

If the selections in figure 2 were in a loop, the interactions right and left would each take place three times before the program terminates. There is also the possible "unfair" computation, in which the simple guard is chosen every time and the program never ends. A discussion of fairness assumptions is given in Section 6.

2.2 Teams and Roles

A team is a concurrent composition of processes (defined as above), and roles, which are formal (parametrized) processes, into which other processes/roles may enrole. Upon enrolment, a process executes the body of the role until termination, when it resumes its previous thread. IP allows for multiple, reentrant enrolment to a given role.

The program prog in Figure 3 includes three teams mainteam, user, and switch_team. The body of a team begins with a (possibly empty) prologue-section. The prologue is a loopless segment executed before the team processes and roles are created. Processes and roles may refer to their team's variables and parameters but not change their values. In prog, mainteam has only a prologue, user has a prologue and a process - P, and switch_team has the two roles - first and second and an empty prologue.

The new command creates a copy of a team along with its processes, and returns a value of type designator (des for short) which identifies the newly created team and which may later be used to enrol into the created roles. The new command may only be activated from team prologues, and thus the team prologues suffice in order to determine the number of copies of each team (and process) that should be created. Every program must have a team named mainteam, and the program starts by executing mainteam’s prologue. If the program is run with any parameters, these are passed to mainteam. The program prog includes one copy of mainteam, one copy of switch_team and 10 copies of user (and of P) which are created recursively and whose designators are entered into the array a.
The team *switch_team* abstracts the two-party activity of switching two values between two processes. Any two processes may *enrole* into the roles *first* and *second* and through them participate indirectly in the *switch* interaction and switch their values. For this purpose, the parameters to *first* and *second* must be passed by reference (the ‘&’ character is added to the parameter declaration).

In *prog*, each process P reaches an NS statement with two *enrolement-guards*. If an enrolement-guard is treated like a simple-guard, then all the guards of the P processes are enabled, and the selections may be made for each process separately. This might lead to a deadlock if, for example, 7 of the P’s select the first guard and enrole into *first* and 3 of the P’s select the second guard and enrole into *second*. If this happens, only 3 pairs may be coordinated to participate in *switch*, and the remaining 4 P’s will wait forever for the role *first* to complete. If, on the other hand, enrolement guards are handled with conflict-propagation, then each enrolement guard’s contribution to the readiness set of P depends on the contents of the role. The first guard of P adds to P’s readiness set, the possibility to participate in the interaction *switch* as role *first*. Likewise, the second guard of P adds to P’s readiness set the possibility to participate in the interaction *switch* as role *second*. This enables the selection to be made in coordination with a number of P processes, and the first guard will be selected for a P process only when a second guard is selected for a matching P process. Thus, with conflict propagation, the program would divide the 10 P processes into 5 pairs (nondeterministically) so that each process knows the id of its partner.

An enrolement statement does not have to be part of a guard. For example a process that reaches a statement - "first@sw(x)" would enrole into the role unconditionally.

### 3 The Implementation Model

IP was implemented on a Meiko machine at the Computer Science faculty in the Technion. This system includes a central machine - "the host" (a Sparc) and a network of over fifty processors - "the nodes" (transputer and i860 chips). A program running on the host may load processes on the host and nodes. More than one process may be loaded on the host and on each node processor, in which case these are executed concurrently with context-switching. However, a set of nodes is allocated to a single program for the entire execution of that program, and this allocation is not dynamic. When all the loaded processes terminate, the control returns to the loading program on the host. The loaded processes may not themselves load additional processes, and thus no dynamic loading of processes is supported. When the loading program on the host terminates, the control returns to the operating system.

The host and node processes of a program (except for the loading process on the host which is suspended while the loaded processes have not all terminated) may all communicate with each other by using synchronous or asynchronous, order-preserving and safe message-passing.

The Meiko system includes a number of compilers for translating programs written in a number of high level languages (including C and Fortran) into the machine code of the host and node processors. Special function libraries are supplied with these compilers to allow the programs to execute the loading and communication actions that were described above.

The IP compiler generates, according to the source IP program, a number of C programs that use these libraries. These programs are then passed to the C compiler which generates programs in machine code that may later be loaded as processes on the Meiko computer, and so simulate the source program.
4 The General Plan of the Implementation of IP

In this section we shall describe the general plan of the IP system that was implemented. First, the specific implementation problems of the treatment of teams, processes and roles will be discussed, as these have a strong influence on the plan of the project.

4.1 Implementation of teams

As mentioned earlier, new teams may be created only from the team prologues, and a team’s prologue must be executed before its processes and roles are created. The team’s processes and roles may then access, but not change, the team’s variables and parameters. Thus, it suffices to execute the team prologues in order to determine how many copies are created of each team and process of the program. Furthermore, the only purpose of these prologues is to create new team copies and to set the values of the variables and parameters that will later be used by the team’s processes and roles. It is therefore possible to first execute all the team prologues separately for this purpose, and then, according to the results of this execution, to execute the program processes and roles. The implementation of IP teams was done according to this scheme:

The IP compiler creates a prologue-program, which consists of the source program stripped of its processes and roles, with only the team-prologues remaining. Each team prologue is translated into a matching procedure in the prologue-program, and each team creation statement (new) is translated into a procedure call to the procedure that matches the created team. The purpose of the prologue-program is to create a team-table, in which each entry describes a created team copy. For this purpose, whenever a prologue-program’s procedure returns, it adds to this table an entry that contains the unique designator of the newly created team and the values of the team’s variables and parameters. The prologue-program also contains a main module that initializes the table and calls the procedure that corresponds to the team mainteam. Before an IP program is run, its prologue-program must first be executed in order to generate the team-table. The loader can then use the team-table in order to know how many copies are required of each program process.

The team-table may also be used at run-time whenever a process or role needs to access one of its team’s variables or parameters. This is accomplished by loading an additional server process - IPNODE along with the program processes. The team-table is also used during the interaction selection phase, because all the potential participants of an interaction should be consulted.

This implementation scheme relies heavily on the static nature of team creation in IP. If processes were allowed to create new teams it would have been impossible to separate the execution of the team-prologues from the rest of the program. In such a case, every dynamic creation of a new team would result in an update of the team-table. This update would have to be known to all the relevant processes. A possible approach would be to first create the new processes and then send the new part of the table to the existing processes that need to communicate with the new ones.

4.2 Implementation of processes and roles

The method chosen for the implementation of processes and roles was to have the compiler generate a separate program for every process and a procedure for every role. Each enrollement statement is translated into a procedure call to the procedure that matches the role being enroled into. Thus, a library is created from all the role procedures, and the process programs
are compiled with this library. This way, a process is linked with all the role procedures that it might call directly or indirectly (through enrolments), and only with these role-procedures.

The loader uses the team-table to know how many copies should be loaded of each process program. Each such loaded copy of a process program is called a basic-process. The basic processes may now run concurrently, and call the role procedures whenever this is needed.

The disadvantages of this approach are that the code that is loaded for each basic-process has to contain the code of all the role procedures that might be called during the execution of the process. Some of these roles may never actually be enroled into. Moreover, if a number of basic processes are loaded on the same processor, it might be possible that duplicate copies of the same role procedures are loaded along with these processes. Also, the role names must be unique within the program (unlike, for example, process names, which must be unique only within their team). The reason for this limitation is that if there were two roles with the same name, say \( R \), then the identity of \( R \) in the enrollement statement \( R@des(...) \) could only be resolved at run time according to the value of the team designator \( des \). This situation must not occur if roles are implemented as procedures, because role names in enrollement statements must be resolved at linking time.

An alternative implementation of roles that solves these problems is to implement roles as separate processes. An enrollement would be implemented by sending a message to this role process, and waiting for a return message from it. Such an approach, while solving the problems mentioned above, creates more serious problems. First, with that solution there cannot be "real" concurrency in concurrent enrollements to the same role from a number of processes. More significantly, the message passing needed to implement enrollement is much more time consuming then the procedure call of the original approach.

4.3 The components in the implementation of IP

Figure 4 shows the general plan of the IP implementation. It includes a compiler (IPC), a loader (IPRUN) and server processes that are loaded with the basic processes (IPHOST, IPNODE, IPPROC, and IPCOORD). The stages in running a program are:

1. A file containing the IP program is passed to the compiler - IPC.
2. IPC creates an archive file which includes a program for each process, a procedure for each role, the prologue-program, and Table files containing various information about the program - such as a table describing the interactions of the program and their participants, and a table that contains information about the external variables that need to be accessed during an interaction by the participants.
3. When the user runs the program, the archive file is passed to the loader IPRUN.
4. IPRUN runs the prologue-program that generates the team-table.
5. IPRUN uses the team table to determine the number of copies of each process program that need to be loaded, checks how many processors are available for the program, and maps processes to these processors. If there are fewer available processors then basic processes, then some processors will be loaded with more than one basic process. IPRUN can now load the server processes described below.
6. After the basic and server processes are done, control returns to the operating system.

The server processes are:
an IP program: \texttt{p.ip}

an archive file: \texttt{p.out}

program p
{
    team t1
    process p1
    process p2
    role r1
    role r2
    team t2
}

Project structure

IPC
a compiler for IP

IPRUN
a loader for IP

The host machine

Team table

Prologue-program

Team prologue file:

Table files:

Process files:
\texttt{p1} \texttt{p2} \texttt{p3}

Role files:
\texttt{r1} \texttt{r2} \texttt{r3}

Input/Output

Loading

Communication

each processor is first loaded by IPRUN and can then communicate with iphost and with the other processors

Figure 4: The Plan of the Implementation
• **IPHOST**: A server process that runs on the host and monitors the status of the basic processors in order to detect termination or deadlock in the program.

• **IPNODE**: A server process that is loaded once on each processor used by the program. It gives the following services:
  - handles the communications needed to send/receive the external variables that are needed from/by the basic processes on the node, when they are participating in interactions.
  - cooperates with **IPHOST** to monitor the status of the basic processors on its node.
  - keeps a copy of the team-table which holds the values of the variables and parameters of the program teams. It can receive requests from the basic processes on the node that need to read these values.

• **IPPROC, IPCOORD**: Each basic process has its own copy of these two servers, that give it interaction-coordination services according to the algorithm of [12], as described later.

5 Computing the readiness-set with Conflict-Propagation

As has been described in the introduction, whenever a process reaches an NS statement, it first needs to compute its readiness-set and only then can the selection be made with the cooperation of other processes that have computed their readiness-set. The readiness-set is the set of the actions that the process is ready to execute next. It remains to define what this readiness is.

Readiness will be defined for processes only, and not directly in terms of roles. Thus, for example, if a process \( P \) enroles into role \( R_1 \) which enroles into \( R_2 \) which is then "ready" to execute the action \( a \), that we shall consider this readiness as \( P \)'s readiness to execute \( a \) indirectly through enrolements. This interpretation is derived from the implementation of roles as procedures in which role actions may be viewed as the actions of the main module that started the series of procedure calls. Thus it is possible for a process \( P \) to be ready to participate in an interaction \( I \) as a role \( R \). When a readiness-set includes an interaction, this entry should also include information that describes the participant that is ready to take part in the interaction. Being ready to participate in an interaction as role \( r_1 \) is different from being ready to participate in it as role \( r_2 \).

If enrolement-guards are treated like simple-guards (that is, there is no Conflict-Propagation), then we use the definition of being ready seen in Figure 5.

Computing the readiness-set according to this definition involves no special implementation problems. The process or role simply needs to compute the boolean expressions of the NS statement and add an action to the readiness set for every readied guard. For example the process \( P_1 \) of Figure 2 will compute the boolean expressions \( R < 3 \), \( L < 3 \) and \( R + L = 0 \) and enter into its readiness the possibility to take part in \( right \) as process \( P \), take part in \( left \) as process \( P \) and to execute the action \( skip \).

However, computing the readiness-set according to this simple approach might lead to problems. What is needed is for the contribution of an enrolement guard to the readiness set to depend on the role being enroled into. There are a number of alternative ways to define this dependency, with varying complexity of implementation. These alternatives, including Direct/Bounded & Full Propagation are discussed in [6], where the concept of Conflict-Propagation is introduced. For example, one alternative is to take the first action of the role
A process $P$ is ready to execute an action $a$ if $P$'s control has reached a command either in $P$ or in a role that $P$ has enrolled into (possibly indirectly) which is:

1. The action $a$.
2. A $NS$ (or $NL$) statement which includes
   (a) A readied simple guarded command "$B \rightarrow A$" (that is, $B$ is true) where $a$ is the first action in $A$.
   (b) A readied interaction guarded command "$B \& a \rightarrow A$" (that is, $a$ is the interaction of the guard and $B$ is true).
   (c) A readied enrollment guarded command "$B \& R \rightarrow A$" (that is, $a$ is the enrollment of the guard and $B$ is true).

The process $P$ is ready to execute $a$ until it starts its next action (not necessarily $a$).

![Figure 5: Definition of ready](image)

as its contribution. This would solve the possible deadlock problem of the program in Figure 3, because each $P$ that reaches the $NS$ statement will compute a readiness-set that includes the possibility to take part in $switch$ as role first and the possibility to take part in $switch$ as role second. Thus, the first guard will be selected for $P$ only if the second guard is selected for some matching $P$. However, such a definition of readiness becomes problematic when the first action of a role is an $NS$ statement, and even more problematic if this $NS$ statement includes a readied enrollment guard. To solve this, one can either prohibit programs from containing roles that start with $NS$ or $NL$ statements (Direct-Propagation) or to prohibit programs from enabling a role to start with a statement that may lead to a direct or indirect recursive enrollment (Bounded-Propagation). Full-Propagation poses no constraints on the program, but may lead to undesirable effects such as an infinite readiness-set.

The Conflict-Propagation that was implemented for IP is a version of Full-Propagation, which is parametric with the parametric bounds $EL$ and $SL$. The definition of readiness remains as above, except for part 2,(c) which is replaced with Figure 6.

![Figure 6: Readiness under conflict propagation](image)

The motivation of looking at paths that include actions beyond the first actions of roles may be demonstrated by Figure 3. If the roles first and second would have included an assignment statement before its participation in $switch$, then looking only at the role's first actions would make the readiness-sets of the $P$'s contain these assignments instead of the interactions and the selection process might again lead to deadlock.
The first constraint on \( \pi \) is needed because a process cannot at the same time be considered ready to take part in one interaction and with another interaction that follows the first on the same path. The second constraint is needed in order to bound infinite paths that include recursive enrollements. The third constraint is needed in order to bound infinite paths that include infinite loop traversals. The motivation for the fourth constraint is to give precedence to interactions over simple actions. This is achieved by extending the path until either an interaction is reached (the first constraint) or until one of the limits is reached. Such precedence is simply a heuristic to encourage maximal information exchange.

These constraints are sufficient to ensure the finiteness of the readiness-set. This may be seen by viewing the paths \( \pi \) (the \textit{Readying Paths}) as paths in an \textit{Enabled Computations Tree} which includes all the enabled computations paths of the process, and in which the \textit{NS} and \textit{NL} statements are represented by vertices that have a branch for each enabled guard. If this tree is "trimmed" so that it includes only the readying paths, then we get a tree of limited depth (because the readying paths are finite by the constraints) in which the leaves constitute the process's readiness-set. The degrees of the vertices of this tree are finite, and so the tree, and hence the readiness-set, is finite by Königs lemma ([13]).

The computation of the readiness-set now reduces to the computation of the "trimmed" Enabled Computations Tree - henceforth the \textit{Readying Computations Tree} or \( RCT \). This can be accomplished according to the following scheme:

The compiler generates the role procedures, such that a role could be executed in three modes: \textbf{normal}, \textbf{search}, or \textbf{follow}. When an enrollement statement is executed the role procedure is called in \textbf{normal} mode in which the role simply executes its code. When a process or role reaches an \textit{NS} or \textit{NL} statement (the \textit{root-statement}) it starts a \textbf{search-phase} in which it computes the \( RCT \) at that point (Figure 7). During the search-phase, the role procedures

1. Create the root for the \( RCT \). This root represents the root-statement for which the readiness-set is computed.

2. For each readied Simple Interaction guard of the root-statement, create a new leaf and connect the leaf to the root.

3. For each readied Enrollement-Guard of the root-statement, create a new vertex - the \textit{enrollement-vertex} that represents the enrollement statement of the guard, connected it to the root. The role of the guard can now be called in search-mode with the enrollement-vertex as a parameter. When a role is running in search-mode, it creates the sub-tree of the \( RCT \) that stems from its enrollement-vertex as follows:
   - When an \textit{NS} or \textit{NL} is encountered while running in search mode, a new \textbf{selection-vertex} is created and is expanded into a sub-tree in a similar way to that described in rules 2 and 3 (thus, rule 3 may be activated recursively a few times).
   - Separate "enrollement" and "step" counters are kept in each new vertex, so that the Enrollement and Step limits can be maintained. When one of these limits is reached, the vertex is made a leaf and is entered into the readiness-set.

![Figure 7: The search phase](image-url)

are called in \textbf{search-mode}. After the search-phase, one of the alternatives represented by the leaves of the \( RCT \) is selected according to process described in the next section. Finally in the \textbf{follow-phase} the readying path that leads to the selected leaf in the \( RCT \) is traversed and the selected action may be executed. During the follow-phase, the role procedures are called in \textbf{follow-mode}.

The left part of Figure 8 shows the \( RCT \) that would result from the search-phase executed
by a process P of Figure 3 when it reaches its NS statement. Now suppose the role second of the team switch_team would be replaced with the following:

```plaintext
role second(int &x2)
{
  [true & switch[x2:=x1] - >skip
   [] true & first@self(x2) - >skip ]
}
```

where `self` is a reserved word of IP, designating the team designator of the team in which the reference to `self` is made. Thus the role second is ready either to participate directly in switch as the role second or indirectly as the role first of its own team. The resulting RCT is shown on the right part of Figure 8.

The search-phase is further complicated by the appearance of input/output statements in the readying-paths. The policy that was implemented was that whenever an input statement is reached during the search-phase, the readying-path is trimmed at that point and a leaf representing the input statement is created, and the statement is entered into the readiness-set. The path must be trimmed in this way, because the rest of the computation may depend on the input. The definition for being ready given above needs to be changed accordingly: the fourth constraint does not apply to input actions and an additional constraint must prohibit π from containing an input action except possibly the last action. The treatment of output statements doesn’t have to be so drastic. When an output statement is encountered in search-mode the search goes on as usual except that the output is not executed (otherwise all the outputs that occur in readying-paths will be made even before the selected leaf is reached in follow-mode).

It may be shown by induction on the depth of the vertices that each vertex of the tree that is created during the search-phase ends a path that begins at the root and that constitutes a prefix of a readying-path as defined above. Also, every leaf of this tree ends a path that begins at the root and that constitutes a complete readying-path. The time complexity of executing the search-phase is like that of making a DFS traversal of the RCT. This has to be multiplied by the maximum number of statements that occur in the program between two statements that receive a vertex in the tree (enrollement, NS , NL and input/output statements). The size of the RCT depends on the Enrollement and Step limits and the maximum number of guards in the NS and NL statements of the program.

The follow-phase: The follow-phase begins with finding the path that begins from the root and ends in the leaf that was selected for execution. At this stage, this path is traversed again by following the path in the tree and executing the actions in it. When an enrollement-vertex is reached the role-procedure is called in follow-mode and is given the enrollement-vertex
as a parameter. It is possible to save time in the follow-phase by letting the vertices of the $RCT$ include the state of the role's variables at the point when the vertex was created during search-mode. This enables a role in follow-mode to skip the intermediate statements between the statements that are represented by the $RCT$ vertices, and only execute the enrolements and output statements of the readying-path. When a leaf is reached in follow-mode, the memory occupied by the $RCT$ may be freed, and the mode is switched back to normal-mode.

Since Conflict-Propagation is obviously more time consuming then treating enrolement-guards as simple guards, the implementation for IP lets the user decide if he needs Conflict-Propagation or not. If Conflict-Propagation is requested then the user must also supply the Enrolement and Step limits.

6 Coordinating Interactions

The second element of the implementation of interactions is coordinating the participating processes after these have computed their readiness-sets. If the readiness-set of a process only includes local actions (actions that are not interactions) then the process can make the selection privately without consulting other processes of the program. This can be achieved, for example, by using a fair random number function to choose the guard to be executed. The fairness of this function ensures that if an infinite number of selections are made from the same readiness-set, then each choice in the selection will eventually be selected for execution.

The possibility of interactions in the readiness-sets requires that the selection be made in coordination with the other processes of the program, because if an interaction is selected for a process, than the same interaction must also be selected for all the other participants of the interaction.

What is needed then is an algorithm for the coordination of first-order interactions (where the participation can be indirect, through roles) in an environment with a constant group of processes. The interactions can have any number of participants whose identity is known in advance. Each process is ready from time to time to execute an action out of a readiness-set of readied actions (which may include readiness to take a local action or readiness to take part in an interaction as a process or a role participant). The process will wait for a selection of a readied action to be made by the coordination algorithm. If an interaction is selected then the same interaction is selected for a matching group of participants for the interaction.

The process is considered ready until it starts the execution of the selected action, and only after its completion can the process begin to be ready for a new set of actions. An interaction is considered coordinated from the time that it was selected for execution by the coordination algorithm, and until at least one if its participants has finished the execution of its part of the interaction.

The coordination algorithm should have the following properties:

- **Synchronization-Safety:** An interaction will be coordinated only when there is a 1-1 match between the set of the participants that were predefined for the interaction and a set of running processes, such that each of these processes is ready to be the participant that it was matched with. Such a group of running processes is called a quorum of the interaction.

- **Exclusion-Safety:** Interactions with non-disjoint quorums cannot be coordinated at the same time (or in other words, a process will not be coordinated for more than one interaction at the same time).
- **Quorum-Liveness**: If at some time a set of processes can be a quorum for an interaction, then one of these processes will execute some action (not necessarily the interaction).

  This property does not seem as necessary as the safety properties presented above, but still it is difficult to guaranty progress without it.

- The coordination algorithm should be able to coordinate concurrently the same interaction with disjoint quorums when these quorums exist.

There are a number of known algorithms for problems with similar specifications. There is for example Bagrodia's algorithm for the implementation of the guarded-command of CSP ([3]), which may be viewed as a coordination algorithm for binary interactions (interactions with exactly two participants) that are also zero-order (the participants are the same for every execution of the interaction). [2], [5] and [4] present algorithms for coordinating multiparty-interactions but the model of the problem or of the implementation environment differs too much from that of IP and the Meiko machine, and most importantly, these algorithms are not intended for first-order interactions. The first (and so far only) algorithm for coordinating first-order multiparty-interactions is the algorithm of [12]. The problem model of [12] also differs from that of IP, but may relatively easily be fitted as needed. Since this algorithm was chosen for the implementation of IP, we shall describe it here briefly:

The algorithm consists of two coordinator server processes - proc and coord attached to each program process. The coordination starts when a program process, say P, sends its readiness-set to its proc. This proc sends the same readiness-set to the coord server of P. This coord now tries to coordinate each interaction in the readiness-set in turn. Each time coord tries to build quorums for the coordinated interaction by capturing the proc servers of the processes in the potential quorums by sending capture requests. When a proc server receives a capture request it may agree to be captured if the interaction is in the readiness-set of its process, or decline if not. If this proc is already captured by some other coord it can also defer its final decision if to agree or not, until its current capturer has either succeeded or failed in coordinating a quorum. proc also has the option to switch its capturer. proc's decision between deferring the request or switching the capturer is made according to the "age" of the competing coords. The age of the servers is determined by maintaining local clocks in these processes in the manner presented in [14].

If coord succeeds in building a quorum for an interaction from the readiness-set, it notifies the proc servers of all the processes in the quorum of its success, and they then pass this selection on to their client processes. These may now execute the selected interaction. If coord fails to build a quorum for every interaction in the readiness-set, then it announces this to the proc server of the same program process. This proc can now only wait to be captured by some other coord that needs it for a quorum. A process can cause at most \(6m\) messages to be sent (by its own servers or by other servers) for each interaction in its readiness-set, where \(m\) is the number of processes that might be ready to participate in the interaction directly or indirectly.

This algorithm corresponds to the specifications given above, and after suitable transformations may be made to suit the model of IP and the implementation environment.

In general, there are a number of fairness requirements that may be thought suitable for interaction coordination algorithms. Unfortunately, these common fairness definitions are inappropriate (by the criteria in [1]) because they relate directly to the execution of interactions, without considering which processes participate.

The problematic situation that we want to avoid is that a process \(P\) is never coordinated to any interaction in its readiness set, while at the same time an interaction in its readiness set
set, say interaction \( I \), is coordinated infinitely often without \( P \) as participant. This is possible when \( P \) is ready to take part in \( I \) indirectly as a role participant, say \( R \), and so other processes besides \( P \) may be willing to take \( P \)'s place and participate as \( R \). The problem results from the possibility of concurrent enrolment in \( I \), in which more than one process may enrol into the same role (directly or indirectly) at the same time.

The new fairness requirement we have adopted may be defined as follows:

| Enrolee-Fairness (EF): | If a process \( P \) is ready to participate in an interaction \( I \) from some moment and forever, and if \( I \) is executed (with or without \( P \)) infinitely often in a computation, then \( P \) will execute some action infinitely often. |

This fairness can be guaranteed by augmenting the \texttt{coord} server to dynamically change the precedence given to which process will fill each role in successive activations of an interaction.

Fairness is also enforced between local guards and interactions. As explained so far, the algorithm assumes that the readiness-set only includes interactions, even though local actions may also be ready to execute. One solution is to treat local actions as one-party interactions, but this would result in too great a preference being given to local actions over interactions. This happens because whenever a \texttt{coord} server would try to "coordinate" a local action, it would immediately succeed and would not try to coordinate the interactions of the readiness set. In fact, if all the processes have local actions in their readiness sets, no interaction would ever be selected by the algorithm. This problem can be fixed by handling the local actions separately from the interactions. For example, the solution that was implemented includes letting the \texttt{proc} server pass only the interactions of the readiness-set to the \texttt{coord} server for coordination. If \texttt{coord} fails in this, \texttt{proc} still lets some other \texttt{coords} try and capture it for interaction quorums, and only when this also fails, is a local action selected at random from the readiness-set.

7 Data transfer

After all the participants in an interaction have computed their readiness-sets (section 5) and have been coordinated to participate in the interaction (section 6), the execution of the interaction reaches its third and final stage: performing the assignments in the interaction's parts. As has been shown in Figure 1, a participant may need to assign an expression that refers to external variables. Furthermore, these references to external variables (like the references to local variables) refer to the values of the variables \textit{before} the participants started executing the assignments.

Action Systems ([2]), which were implemented on a bus architecture, implemented this by having each participant broadcast the values of its local variables, and thus allow every participant to pick up the externals it needs. Such a solution would obviously be very costly on the message-passing transputer model described in section 4, in which each such broadcast would be implemented by sending the local values to all the processes of the program. Here we only have messages from the sending participant with the external values that the receiving participant needs for its assignments.

These messages are sent and received through the \texttt{IPNODE} server processes (of which there is one copy running on each processor). \texttt{IPNODE} packs and sends together messages that are to be sent to participants running on the same destination processor. This saves on the number of inter-processor messages at the expense of more intra-processor messages. Suppose the interaction \( I \) has \( P_I \) participants running on \( N_I \) nodes. The external data transfer for \( I \) would require \( O(N_I^2) \) inter-processor messages between the \texttt{IPNODE} servers.
and $O(P_1 \times N_I)$ intra-processor messages (implemented using shared memory) between the IPNODE servers and the participants (each participant might be required to respond to at most $N_I$ requests for externals).

While it is possible for the compiler to compute the identity of the externals that need to be received by a process that reaches an interaction part, it is impossible for it to compute in advance the identity of the local values that a process will need to send to the other participants, because these differ according to the interaction parts that were reached by the other participants. For example, if in Figure 1 the two parts of cycle in P2 were guards in a guarded command of P2, then the compiler could compute that when P1 executes the first part of cycle it needs to receive X2 from P2, but the compiler cannot tell if P1 needs to send X1 to P2 or not. To enable each IPNODE to send the correct externals, the compiler computes a table which identifies the externals that need to be received by each participant from each other participant, depending on the part executed by the receiver. This table is supplied to the IPNODE servers at loading time. When the coordination stage of the execution of an interaction is completed, all the relevant IPNODE servers receive a message that identifies the interaction parts to be executed by each participant. IPNODE may thus refer to the correct entries in the table and send the externals as needed.

The implementation of the external data transfer stage is further complicated if external array references are allowed. For example, if the index of an external array refers to external variables, then another "round" of communication is needed - the first to compute the index and the second to send the actual array member. If the index of an external array is nested (i.e., is allowed to refer itself to external arrays), then the number of similar additional rounds of communication depends on the nesting depth. This last complication was avoided in the implementation of IP by allowing only simple array indexing within interaction assignments, that is, by forbidding nested array references.

8 Conclusions

In this paper we have described some of the main features in a compiler system for IP, and indicated some solutions to implementation issues. In particular, a scheduling algorithm must be used to select enabled interactions in a way which will avoid deadlock, and exploit the transputer architecture. Conflict propagation is used to determine the readiness sets. Moreover, roles from relevant teams must be distributed among the processor nodes, and a run-time environment must be provided that allows executing the translated code, and in particular sending messages that implement data transfer needed for multiparty interactions.

Additional optimizations in this compiler should allow high-level programming that is efficiently implemented and leaves most communication problems to the compiler.

Acknowledgment: The work of Allon Adir and Nissim Francez was partially funded by a grant from the Israeli Academy of Science (Basic Research program). Nissim Francez and Shmuel Katz were also partially supported by the Fund for the Promotion of Research at the Technion.

References


