Distributed Declarative Systems as Parts of Cooperating Software Environments *

G. Czajkowski, A. Uszok, K. Zieliński
Institute of Computer Science, University of Mining and Metallurgy
al. Mickiewicza 30, Kraków, Poland
e-mail: {grzes, uszok, kz}@ics.agh.edu.pl

March 1994

The paper addresses the problem of using software environments allowing distributed declarative programming, as components of an open system. This enables declarative programs to call services provided by other languages, possibly with different computational models, like object-oriented, or imperative. On the other hand, a declarative system may export certain services, thus becoming a server in a set of cooperating systems. Such gluing together software environments, as opposed to connecting applications, has its advantages and disadvantages which we present together with describing such a hybrid system called Interoperability Architecture (IA), parts of which are Strand and PCN. Then we concentrate on the way programming in the languages mentioned above is influenced. Problems and solutions that exist when declarative systems are added to IA are presented together with some implementation details. The paper is concluded with one bigger example.

1 Introduction

Today's information systems consist of a vast network of distributed computing and information resources. With growing number of such systems, there is an increasing need for technology to integrate them, creating an integrated distributed computing system.

While global interoperability may be achieved by the universal adoption of a particular computational model [3], it is likely that no one model will achieve universal acceptance. Hence, the best achievable interoperability solution is to create mappings between the features of the various models and provide uniform platform for different computing paradigms [7, 12, 8, 2], such as imperative, functional or logic programming. Previouse work in this field have been mainly concentrated on interoperability between programs written in different sequential languages i.e Pascal, C, Lisp and Icon [7], Ada, Lisp and C++ [12] or C, Lisp and Prolog [2].

In this paper we would like to present our view on how distributed declarative systems, which emerged in last few years, might become members of cooperating software environments. After introducing a notion of Interoperability Architecture (called thereafter IA), details of which are presented in [13], we present the mechanisms that should be added to declarative systems that allow these systems to become active parts in a set of programming systems. However, during the design we paid attention to make all solutions as general as possible, so that any

*This work was sponsored by Polish Research Council (KBN) under grant no.: 8 S503 015 06.

other declarative system may be added to IA using methods described in this paper with the minimal (possibly none) amount of change to our implementation. We cannot assess generality of presented methods, but we are certain that the addition to IA of a system belonging to the family of parallel, logic programming, committed choice languages that allow calling C code can be based to a considerable extent on solutions described in this paper.

Finally, some implementation details are discussed, together with one bigger example, in which Strand [6, 9] and PCN [4, 5] cooperates with ANSA [1].

1.1 Overview of Interoperability Architecture

Interoperability Architecture (called IA) is built around a family of distributed languages and software tools integrated by the specialized component called Interoperability Server (IS) (see Fig. 1). Every environment–member of IA communicates with IS using a *stub* – a part of IS, responsible for data translation and transport between IS and that environment. Each stub is written partially in the environment which it represents in IS and partially in the system in which IS is written (initially we used ANSA; later IS was implemented in Orbix, an implementation of CORBA). Any environment can register its services for use by other IA environments and call services registered by those environments.

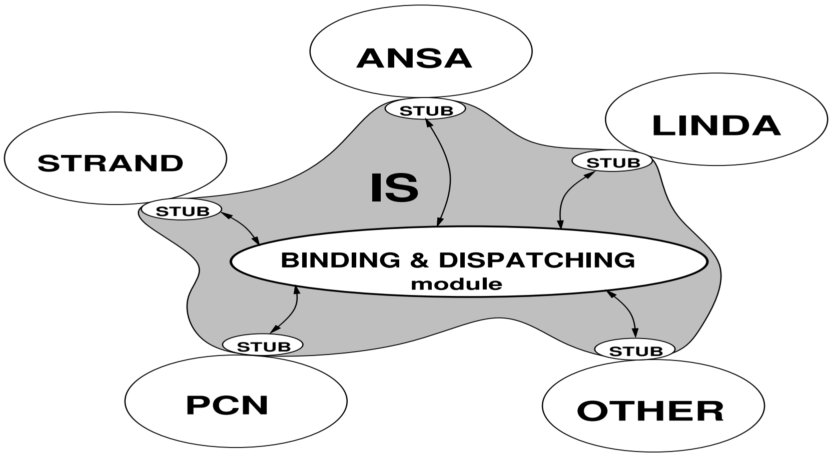

Figure 1: Interoperability Architecture with Strand, PCN, ANSA and Linda environments

IA offers an uniform platform for the application software development. The main effort is to allow interoperability among diverse distributed systems without creating extra complexity. IA provides functionality addressing the problem of cooperation of various software environments or integration of software components. They can be developed as separate programs in any language supported by IA. It provides mechanisms which make some distributed environments open what means that a new application program may cooperate with programs developed earlier, possibly after a minimum amount of changes.

Similarly, we try to design IS under the condition that it should preserve the computational models of linked environments and be transparent for a programmer wherever it is possible. The very important design issue concerned data types.

1.2 Motivation of this study

There are basically two approaches to integrating software: on the application and source code level (the latter approach is adopted by IA). Their features are in general complementary, that is, the advantages of integrating applications are disadvantages from the viewpoint of integration of source code and vice versa. The detailed discussion can be found in [13]; here we will briefly state why we think the second approach is attractive and what are the benefits of using declarative languages in such a mix of systems:

- The IA approach gives us the opportunity to integrate programs at the source level, one may use several different environments to create his or her application, taking advantage of the features available in different computational models and using systems best suited for a particular job.
- It provides support for software reuse, because the object/procedure wrapping techniques may be used without limitation.
- It solves out in a uniform way a problem of hybrid application implementation that combine different computational models. This leads to the elimination of error prone *ad hoc* designed interfaces between application written in different languages. The performance may benefit from using fine-grained parallelism if only component languages support it.
- It provides mechanisms to open a closed software environment that is it makes a cooperation possible via dynamic binding of the programs written in the same or different languages.

Having in mind what has been said above, one can clearly see the usefulness of having the declarative systems cooperate with the others. Their disadvantages become less painful because for things not suitable for being solved in Strand, like numerical processing, may be programmed in other languages. Also, one may reuse existing code practically without limitations, exporting services for use in IA (as shown later). For example one does not have to rewrite a long Linda code to PCN – it suffices to place a call to a service provided by Linda.

On the other hand, systems making up IA will benefit from having declarative languages among them. They can provide easy solutions to for example symbolic processing and decision problems. Also, in a distributed processing, the notion of *virtual topologies*, present for example in Strand and PCN, might prove attractive for other systems, because of easiness of mapping processes to hardware reflects the nature of the problem.

1.3 Generality of proposed solutions

All solutions and conventions adopted during our project can be divided roughly into two groups:

- Solutions valid for any IA member environment, regardless of the computational model,
- Solutions specific for a given computational model.

Below, we briefly describe our generalized solutions concerning data types and error handling for all systems in IA, regardless of their computational model. Others aspects of entering distributed declarative system in IA which are different with regard to object-oriented [11], imperative or functional systems are covered in the next part of this paper. It includes bootstrapping a system in IA, registration of services, calling procedures in different systems and being a server in IA.

Data representation and translation in IA

At the risk of being boring, we notice that different languages have different data types. When linking N environments, the question is: should we make it possible for all languages involved to use all data types they have and translate them as necessary or intersect the sets of all possible types and supply operations on these types only. The second approach has two strongly tempting features, which made us decide on taking it: it is easy to program and fast in use. Besides, it frees us from coping with some problematic questions, e.g: how is the pointer to be represented in PCN ? Or, similarly, in some implementations of Linda, tuple space is an ordinary data structure; one can have an array of tuple spaces. How is that to translate ?

We have decided to use types that are common to all languages we currently use, not only declarative (it is difficult to imagine a language that would not provide them): integers, reals, strings and sequences of these types. Providing for examples structures would be very easy if IA were to connect only declarative systems, but in case of object-oriented imperative systems, like ANSA, one would have been forced to provide descriptions for every possibly used type of structure in the interface definition language. Encoding and decoding of these types are fast operations and we think all types of process conversation/interaction can be expressed using them, although sometimes with a little bit more difficulty as when using more abstract types.

Error and exception handling in IA

There can be different reasons for errors and exceptions in IA. Most of them origin from the fact that computations are carried out in a distributed system, from which the well-known disadvantages are inherited.

Let us assume that procedure P_1 in an environment E_1 calls some service P_2 supplied by the system E_2. First of all, E_2 may crash. To detect such situations, the server linking environments maintains a *keep–alive* protocol. If E_2 does not respond for a certain period of time, it is considered to be down. If E_2 has crashed part way through the call P_2, the appropriate error code is returned to P_1. If E_2 is rebooted later, the calls that were prematurely ended because of the crash are not automatically called once again. It is the responsibility of the caller to take care of getting results right (possibly calling the required service more than once) in the presence of crashes. Thus, IA offers *at most once* [10] semantics. If E_2 is functioning correctly, the possible source of errors may be the called service, P_2. It may be deregistered (or not yet registered) at the moment of call, it may expect different number and types of arguments than supplied or it may return result of type different from the one caller expects (this errors may be common when programming in IA using typeless languages and strongly typed languages together).

All of these errors are reported to the caller, using the most appropriate error reporting mechanism available in a given environment. We assume that IS was written in such a way that it does not introduce its own errors (we did our best to make this assumption a fact). Finally, we would like to point out that if the called procedure does not return for a very long time but its environment has not crashed it is assumed that everything is fine, that is, the computations run the way they should. We do not put an upper time limit on a call execution.

2 Functionality of IS interface

When considering adding a declarative system to IA we assume its following features: it should possess stream-based communication facilities and the ability to integrate calls to C code into its programs. The present implementation is based on UNIX BSD systems. The socket network interface is used (UDP protocol). It should also be possible to build the handler for at least one of the four signals: SIGIO, SIGALRM, SIGUSR1 or SIGUSR2. The last and somewhat weaker requirement is that the system under consideration be distributed.

In this section, we enumerate extensions added to distributed declarative systems for use by the programmer that allow them to cooperate with other environments integrated in IA. In our opinion they are valid for the wide class of declarative systems – here we require only that processes communicate using streams (later, when discussing implementation details we add one more requirement – the ability to link in C code).

As said earlier, the main design issue is to achieve transparency wherever possible. However, we do not try to achieve *total transparency*, because that would be equal to the fact that the programmer is completely unaware of other environments. Clearly, one must be provided with the appropriate constructs that allow:

- bootstrapping an environment in IA,
- communication,
- calling the services provided by other systems,
- being informed of errors,
- registering his or her own procedures for use by other systems.

Below we address each of the five itemized issues separately. Later we present the use of these extensions and example programs written in Strand and show how mechanisms given to the programmer are implemented.

2.1 System bootstrapping

In order to become a part of IA, an environment must contact IS. This is done using a configuration file. It contains the machine and port number of IS. After having read that file, an environment sends information about itself to IS. That information contains a machine name, port number and the code identifying the system (e.g. PCN, Strand, etc.). All the above mentioned activities are accomplished by a process *start_work/3*.

2.2 Communication mechanism

In addition to bootstrapping the system in IA, *start_work(IStream, OStream, Status)* provides the streams which are used to transmit messages between the system and IA. The functions of arguments are as follows:

- *IStream* – this stream contains a part of information sent from IA to the declarative system, that is those requests that have been registered in the *conversation mode* (described later). This stream may be monitored but not modified by the programmer,
- *OStream* – this stream is generated by the programmer and consists of requests to services provided by environments grouped in IA and procedure registration information,
- *Status* – informs of successful bootstrapping and setting up communication channels.

2.3 Calling foreign services

The stream *OStream* (the second argument of *start_work*) is used to call foreign services. We have adopted the convention that every call to an IA service has at least one parameter and that the last argument will contain the return value or, in case of failure, an information about error. As explained before, parameters and results of calls may be only of simple types and list of these types.

2.4 Error reporting

Dealing with errors in the general case (i.e. for any system in IA) was presented earlier and the solutions adopted apply equally well in the case of declarative systems; here we will only add that when an error has occurred a tuple { *"error", ErrorCode* } is generated. The tuple is prohibited as the return value of a normal remote call – its very presence signifies an error and at the same time we do not have to reserve some of allowed return values for error reporting.

2.5 Registering services

Any procedure that may be remotely called by an IA environment must be registered beforehand. We introduce two modes of registration in declarative systems added to IA: *system* and *conversation*. The first is used when there no connection between subsequent calls to the same service and when the programmer does not have to be aware of the call. It provides the transformational, or functional, services: parameters are transformed and results returned without any communication with other processes in the declarative system. It can be said that system mode is used to build stateless servers. That is a consequence of the fact that there is no global data in Strand or PCN and we do not introduce such a notion to these languages, sticking to the decision not to change the computational model.

On the other hand, conversation mode is used when the programmer wants to monitor incoming calls to a given procedure or when subsequent invocations of the service should share some information. The existence of these two modes is particular to declarative systems[1], because of the lack of global variables.

These modes determine how Strand and PCN handle incoming requests to call registered services and are important only within these systems; IA gets one registration message type, regardless of how a service is registered by a programmer in Strand or PCN and a client of a service has no way of knowing under what mode they were registered.

3 Programming interface

In this section we elaborate on the topics discussed so far and present how the mechanisms described may be used in Strand. We stress that adopted ideas are general – the programs presented, modified only because of syntax differences, were run for example in the PCN system.

3.1 Bootstrapping

The process *start_work/3* is defined in the module called *ia*. The following call will bootstrap Strand in IA:

[1]In fact, it may also be necessary when incorporating functional languages into IA. So far, we have not don such experiments.

```
go :- ia:start_work(IStream, OStream, Status),
        ... % rest of the program
```

The *Status* variable should be examined for an error code.

3.2 Calling foreign services

If for example one of IA processes exports the *cube* service, Strand user may compute *7* to the third power like this:

```
OStream := [ cube(7, Result) | OStream1 ].
```

Thus, invoking a foreign procedure is done using an *incomplete messages* technique. *OStream* is an ordinary list and can be merged or (in PCN) distributed among processes. Although there exists a way to check what system provides a certain service, in general the calling process should not care; it may happen that Strand process will call some other Strand service and it is perfectly compatible with the idea of IA.

3.3 Registering services

In order to allow other processes to call their procedures, Strand and PCN must register them in IS. This is done by sending a special message { *1, Module, CallName* } or { *2, CallName* } on the *OStream*. *Module* and *CallName* are strings identifying the registered call. Integer constants, *1* or *2*, determine the mode in which the service is to be registered.

System mode

Strand or PCN may export[2] their procedures using system mode. The request is passed directly for execution to the declarative system, which is completely responsible for the execution and there is no possibility of communication between the process servicing the request and other processes in the declarative system. It means that if some system wants a Strand program to execute *fib(7, X)* and the procedure *fib/2* is registered for use, the module *M* containing the appropriate call will be found and the Strand system routine *exec({ M, CallName, _ }, _)* will be called. Similarly, in PCN a metacall facility would be used. If a service has been registered using system mode, there is no communication possible between its invocation and any other processes in a process pool.

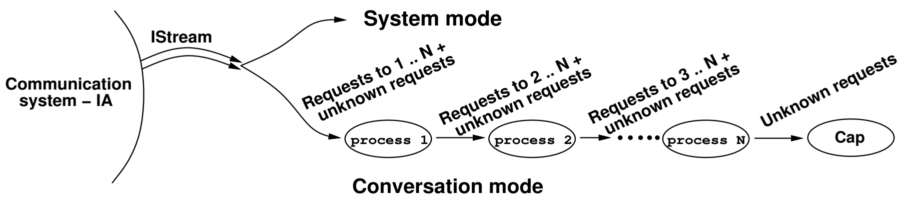

Figure 2: Internal handling of requests to services registered in different modes in Strand and PCN.

---

[2] The use of the word *export* should not be confused with exporting procedures by modules in PCN and Strand systems. Here we use this word to denote providing a service for other IA entities.

Conversation mode

Conversation mode enables foreign processes to communicate with Strand and PCN processes that are currently being computed. If a *fib/2* process is registered in this mode, the request *fib(7, X)* is passed on the *IStream* to the process waiting for such messages. For example, the Strand process providing computing the Fibonacci numbers for IA systems and managing statistical data concerning foreign calls may be defined like this:

```
go(StatQueries) :-
  start_work(IStream, OStream, _),
  OStream := [ { 2, fib } | _ ], % conversation mode registration
  merger([ merge(IStream), merge(StatQueries) ], ToFibServer ),
  fib_server(ToFibServer, 0, NotToFibServer),
  cap(NotToFibServer).

fib_server([ get_stats(S) | Requests ], Calls, NTFS) :-
  S := Calls,
  fib_server(Requests, Calls, NTFS).
fib_server([ fib(X, Y) | Requests ], Calls, NTFS) :-
  fib(X, Y),
  Calls1 is Calls + 1,
  fib_server(Requests, Calls1, NTFS).
fib_server([ X | Requests ], Calls, NTFS) :- otherwise |
  NTFS := [ X | NTFS1 ],
  fib_server(Requests, Calls, NTFS1).
```

The *fib_server/3* process filters out requests to itself from *IStream*; requests that are not meant for it are passed further on via a *NotToFibServer* stream. It is the programmer's responsibility to establish such a chain of processes, like shown in Fig. 2. A special *cap/1* process is provided, which should be used as a terminator of the list of processes. It extracts the last argument from the request tuple and assigns it an error code. As is clear now , IA is not aware (and, in fact, should not be) of the mode in which a Strand or PCN service is registered. This information is important only in these systems and is a consequence of their computational models, namely, lack of global information.

4 Implementation and structure of declarative environment interface

This section will describe how the operations added to declarative systems, enabling those systems to become parts of IA, were implemented. To a large extent, this description is an answer to the question: what happens when a user calls *start_work/3* process ? In previous sections we stressed generality of some solutions; in this chapter, we present the Strand implementation, but it can be adopted equally well for other declarative systems. For example, our addition of PCN to IA was based to a very large extent on our work with Strand.

There are two levels of implementation and general structure of the interfacing stub:

- a higher level, consisting of processes defined in the linked system. These processes maintain dictionaries of variables – return arguments from foreign calls that have not returned yet, manage modules (necessary for system mode registration) and provide a simple form of flow-control. This level is universal for distributed declarative systems.

- a lower level, dealing with bootstrapping, network transmission, connection control, data types encoding and decoding. The structure and implementation of this level will differ, depending on the environment.

4.1 Stub architecture – higher level

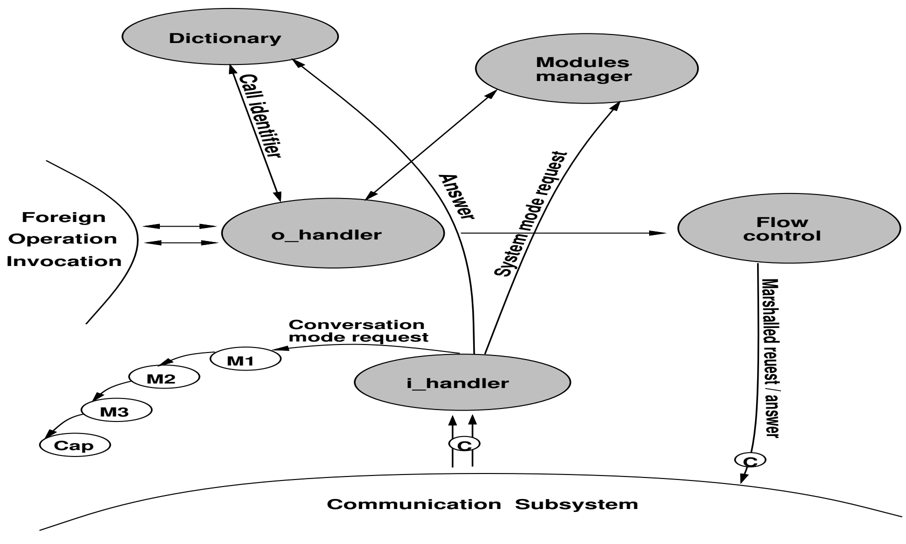

Figure 3: Stub's structure and scheme of activity

Management of connection between Strand and other processes is maintained using a set of processes: *i_handler*, *o_handler*, *dictionary*, *mod_manager* and *flow_control*, as depicted in Fig. 3. They will be described here; next section will present procedures written in C. The points of interface between Strand and C procedures are also marked in Fig. 3.

Process *o_handler* is responsible for managing user requests passed to *OStream*. Process *i_handler* handles requests obtained from IA processes. The task of *mod_manager* is to maintain an information about modules in which services provided by Strand and registered in a system mode are defined. Information about all requests from Strand to IA that have not returned yet is managed by *dictionary*. This data is used when a call returns, in order to pair the result (possibly indicating an error) and Strand variable waiting for it. Next we present in detail *o_handler* process; the role of *flow_control* will be presented later.

Messages from Strand to IA

There are four types of messages sent to *o_handler*, and thus further to IA, as shown in the code below.

```
o_handler([ { 1, Name, Mod } | In ], ToDict, RegisterRequests, ToFlowControl) :-
  RegisterRequests := [ { register, Name, Mod } | RR ],
  ToFlowControl := [ { register, Name } | TFL ],
  o_handler(In, ToDict, RR, TFL).
o_handler([ { 2, Name } | In ], ToDict, RegisterRequests, ToFlowControl) :-
  ToFlowControl := [ { register, Name } | TFL ],
  o_handler(In, ToDict, RegisterRequests, TFL).
```

```
o_handler([ { 3, Name, Mod } | In ], ToDict, RegisterRequests, ToFlowControl) :-
  RegisterRequests := [ { deregister, Name, Mod } | RR ],
  ToFlowControl := [ { deregister, Name } | TFL ],
  o_handler(In, ToDict, RR, TFL).
o_handler([ ToIARequest | In ], ToDict, RegisterRequests, ToFlowControl) :-
  otherwise |
  arity(ToIARequest, Arity),
  get_arg(Arity, ToIARequest, RetArg), % last argument is return argument
  ToDict := [ { insert, RetArg, RetArgNum, Done } | ToDict1],
  send_request(Done, ToIARequest, Arity, RetArgNum, TS),
  ToFlowControl := [ merge(TS) | TFL ],
  o_handler(In, ToDict1, RegisterRequests, TFL).
```

The first two are used to register a service in a system and conversation mode, respectively. This information is passed on to *flow_control*. In a system mode registration, an entry containing a name and module of service sent on the *RegisterRequests* channel to *mod_manager*. The third kind of message causes deregistration of a procedure. Regardless of type of registration, this message is passed to both *flow_control* (and further to IA, as described soon) and to *mod_manager*. If *mod_manager* does not find an appropriate registration information, it discards the message, assuming the conversation mode of the deregistered service. Finally, the last type of message may be a request to IA. It is a tuple, with all but last argument bound. In order to ensure that this is really the case, *send_request* process is used. It waits until the above condition is met and creates the new message, passed directly to *flow_control*, consisting the request header, the request number and all but last argument. This last argument is stored by *dictionary*, as described previously.

Sending requests to IA – *flow_control*

Two streams are arguments to *flow_control*. The first of them contains messages that are to be sent to IA: registration and deregistration information, answers to IA requests and requests to IA. It also maintains a count of requests, providing a unique integer identifier for each Strand request to perform some service registered in IA (second clause). The actions associated with these messages, i.e. sending appropriate messages over the network to IS, are accomplished by procedures written in C (distinguished in the code by a prefix *c_*).

```
flow_control([ { answer, N, A } | InStream ], [ [] | GoStream ], Count) :-
  c_send_answer(N, A, _),
  flow_control(InStream, GoStream, Count).
flow_control([ { request, N, Call } | InStream ], [ [] | GoStream ], Count) :-
  N := Count,
  Count1 is Count + 1,
  c_call_foreign_proc(Call, Count, _),
  flow_control(InStream, GoStream, Count1).
flow_control([ { register, Name } | InStream ], [ [] | GoStream ], Count) :-
  c_register(Name, _),
  flow_control(InStream, GoStream, Count).
flow_control([ { deregister, Name } | InStream ], [ [] | GoStream ], Count) :-
  c_deregister(Name, _),
  flow_control(InStream, GoStream, Count).
```

The second stream is built by C procedures. An empty list is added to it after receiving an acknowledgment from IS. Thus, depending on an initial number of empty lists contained in *GoStream*, the number of messages sent but not acknowledged so far may be limited.

4.2 Stub architecture – lower level

This section describes the part of the stub which depends on the declarative system's implementation, representation of its data structures, ability to link C code and operating system dependencies. The lower level of our implementation concerns translation of data types into network format, maintaining the connection between IS and the given environment, transmitting and receiving data from IS.

In Strand implementation, we have taken advantage of excellent capabilities of this system allowing integration of foreign code. Consequently, all procedures making up the lower level were implemented directly in C. In particular, any incoming data causes generation of a signal (SIGIO). Its handler decodes the message, sends (if necessary) an acknowledgment and builds an appropriate Strand data structure adding it to *IStream*.

On the other hand, in PCN it is impossible (at least in the current version) to pass tuples to C code and to build lists inside that code, so the C code can not send any messages to PCN processes, as opposed to Strand. In order to overcome this difficulty, we have decided to add a special PCN process to call C routine *c_read()*, reading from the socket (this reading is blocking, as opposed to the asynchronous I/O in case of Strand). After decoding the data received from *c_read()* and the appropriate message is sent on the *IStream* channel. Because reading from socket is blocking, it makes the entire PCN virtual processor, which is in fact a Unix process, blocked. In order to enable other PCN processes to compute, *start_work()* is started on a dedicated virtual machine. Thus, the main cost of adding PCN to IA is setting an additional virtual machine

5 Example

The example presented will concern computing a colour bitmap of the Mandelbrot fractal. A Strand program shown may be thought of as a fractal server: it accepts requests to compute the fractal and returns a name of the file in which the bitmap is stored. The program takes advantage of all discussed features: system and conversation mode registration, calling of IA procedures and error handling.

5.1 Description of an algorithm

A fractal image is defined by the width and height of a picture. After receiving a request to compute a fractal of width W and height H the Strand server checks if it has already been computed. If so, the filename associated with $\{ W, H \}$ pair is sent back to the IA caller. In the opposite case, Strand invokes IA operation (supplied by ANSA, a system shortly described in a next subsection) which creates as many workers as there are available machines in a network. Each of these workers is assigned an initial task. After completion of a task, a worker returns results to Strand and asks for another task, until there is no more work. A task code consists of a coordinates of a first point in a pack of points (here it is equal to 1000).

Processes carrying out the task of actual computing a bitmap are defined in ANSA [1], a system which provides uniform platform for building object-oriented distributed applications. Application in this system consist of some number of cooperating independent objects (called *capsules*) written in the extended C language. Interaction between these objects is supported by special stub procedures in C. These procedures are generated according to interfaces, described in the Interface Definition Language, specific to ANSA.

We present this short description of ANSA in order to show how diverse systems may cooperate in IA. Computational models are totally different for Strand and ANSA; also, different are areas of application of these languages. Although almost all programming environments on the market today are dubbed "general purpose", for each of them a problem can be found for which that particular environment is rather poorly suited. Their deficiencies may be compensated for by becoming a member of IA. The programmer will be able to choose the best tool for writing a single *procedure* and this choice will (ideally) not affect the languages in which other procedures will be implemented.

5.2 Interaction of the components

This example uses a model of processors farm for the calculation of a fractal bitmap. In this paragraph the interaction of components which execute in different environments (in this case Strand and ANSA) is described (Fig. 4).

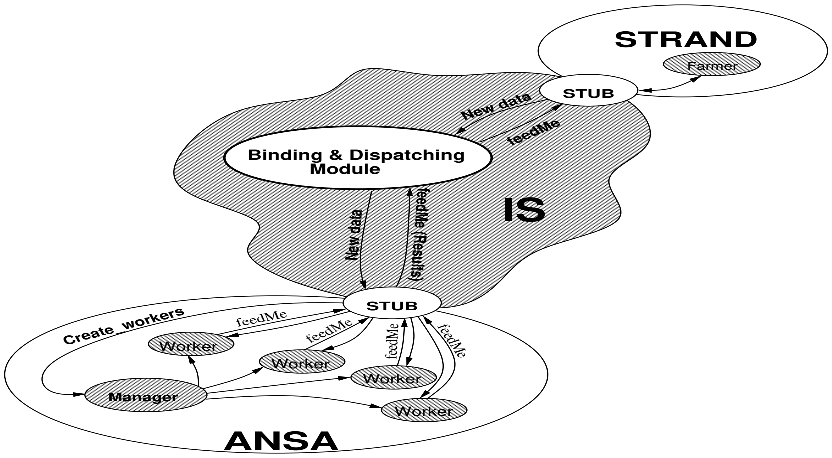

Figure 4: Cooperation of IA's elements.

A *farmer*, written in Strand, is responsible for: initialization of the computation, sending tasks to the *worker*s and receiving from them the calculated parts of a fractal bitmap. After receiving all parts of this bitmap *farmer* writes a complete bitmap to a file. The *worker* processes are written in C and executed in ANSA. If *farmer* is to cooperate with ANSA than IS module which is responsible for binding and dispatching should be run beforehand. Services offered by ANSA and necessary for proper execution of this computation should be registered in this module. In this case it is a function *startworkers* provided by the special ANSA capsule *manager*. Also, services provided by the Strand system should be registered in IS module. In the example this is an operation *feedMe* which returns a computed part of bitmap and asks for new input data.

The output stream from *farmer* is processed by IS which directs it to the appropriate ANSA's capsules. In particular if IS receives request to create *worker*s than it invokes the operation *startworkers* provided by the capsule *manager*. This capsule uses standard ANSA mechanisms to create *worker*s on all nodes belonging to a system.

5.3 Computing colour bitmap

Here we describe Strand programs. First of them starts the fractal service. Initially, *start_work* is called which is assumed to bootstrap Strand system in IA correctly, so no error handlers are necessary (line 1). The process *fractal_compute* is registered in a in a conversation mode (2) It will be called by IA processes. Another service provided by Strand, *feedMe*, registered also in a conversation mode (3), will be used by IA processes-workers, which carry out the actual bitmap computing. These IA processes are started by Strand, when it turns out that the fractal of requested size has not yet been computed. Thus, after Strand is contacted to compute the fractal it requests some processes in IA to start computing the bitmap. These processes return partial results to Strand, asking for another piece of work.

A request to to provide a list of all created will be executed by a *history* process (4), defined in a module *h_mod*. All *fractal_compute* and *feedMe* requests are routed on the IStream to the *compute/4* process (5). Finally, *cap/1* process, as discussed earlier, captures all bad requests.

```
start :-  ia:start_work(IStream, OStream, _),         % (1)
          OStream := [ { 2, fractal_compute },        % (2)
                       { 2, feedMe },                 % (3)
                       { 1, h_mod, history } ],       % (4)
  compute([], IStream, OStream, BadRequests, 0),      % (5).
  ia:cap(BadRequests).
```

A *history/1* process is extremely simple – its actions are the same as calling *ls datfile** on the Strand current directory, so we will not present its definition here. An example call from PCN would look like this:

```
get_fractal_history(History)
{; ia:start_work(IStream, OStream, _), IStream = [history(History)] }
```

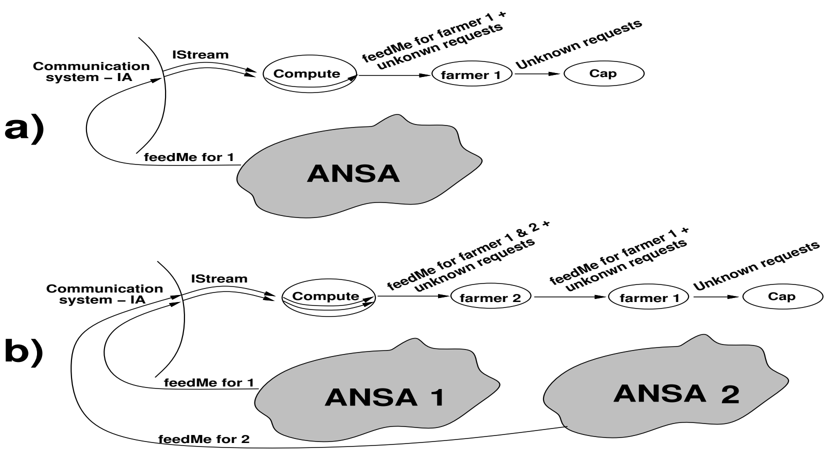

Figure 5: Starting a new computation of the fractal. Figure A shows the situation when one fractal is being computed. Figure B presents the placement of processes after adding one more *farmer*.

The actual job of computing the bitmap is carried out by *compute/4*. Its first argument is a synchronizing parameter. Second is a list of incoming requests. *OStream* is used to init

ANSA procedures that compute bitmap. Finally, all requests that *compute/4* is not prepared to handle are sent for further processing on the *PassOn* stream. These requests which *compute/4* is unable to handle are all *feedMe* calls. Every started fractal computation will be assigned a unique number (*ConnNum*). Each *feedMe* request carries this number so that requests from different computations of fractal executed in parallel with each other can be safely multiplexed on the *IStream*. Figure 5 how the new computation of fractal is started.

```
compute([ fractal_compute(W, H, FileName) | Rest ], OStream, PassOn, ConnNum) :-
  make_fname(W, H, FName), % FName == "data" ++ itoa(W) ++ itoa(H)
  find_cached_result(FName, Found),
  decide(Found, W, H, FName, FileName, Rest, OStream, PassOn, ConnNum).
compute([ H | Rest ], OStream, PassOn, ConnNum) :- otherwise |
  PassOn := [ H | PassOn1 ], compute(Rest, OStream, PassOn1, ConnNum).

decide(yes, _, _, FName, FileName, IStream, OStream, PassOn, ConnNum) :-
  FileName := FName,
  compute(IStream, OStream, PassOn, ConnNum).
decide(no, W, H, FName, FileName, IStream, OStream, PassOn, ConnNum) :-
  OStream := [ startworkers(W, H, 1000, Workers, ConnNum) | OStream1 ],
  CurPoint is Workers * 1000, Total is W * H,
  farmer(PassOn1, PassOn, {FName,FileName}, CurPoint, Total, Workers, [], ConnNum),
  ConnNum1 is ConnNum + 1,
  compute(IStream, OStream1, PassOn1, ConnNum1).

farmer(IStream, IStream1, X, CurPoint, Total, Workers, Result, ConnNum) :-
  CurPoint >= Total |
  finish_job(IStream, IStream1, X, Workers, Result, ConnNum).
farmer([ feedMe(StartPoint, ComputedPack, Work, ConnNum) | IStream ], IStream1, X,
  CurPoint, Total, Workers, Result, ConnNum) :- CurPoint < Total |
  Work := CurPoint,  CurPoint1 is CurPoint + 1000,
  Result1 := [ { StartPoint, ComputedPack } | Result ],
  farmer(IStream, IStream1, X, CurPoint1, Total, Workers, Result1, ConnNum).
farmer([ H | IStream ], IStream1, X, CP, T, W, Res) :- otherwise |
  IStream1 := [ H | IStream2 ], farmer(IStream, IStream2, X, CP, T, W, Res).
```

6 Evaluation of performance and overhead

In this section we present some results concerning the IA performance evaluation. The values presented concern the Fibonacci numbers computation; they are summarized in the table below. This example was chosen because it is both simple and well illustrates possible performance gains for some classes of problems. A Fibonacci number was computed ten times and the value ot the tenth computation was returned to the caller. The table compares the case when Strand did all the computations (column 2), when only ANSA computed the numbers (3) and when Strand called IA, which in turn called ANSA (4). ANSA server was located on different machine than Strand caller. We used SPARC2 machines, connected by Ethernet. These results show that systems like Strand may benefit from being in IA. For some computational tasks the time shrinks by orders of magnitude, even when taking into account communication overhead.

Argument	Calculation time [ms]			Argument	Calculation time [ms]		
	Strand	ANSA	IA		Strand	ANSA	IA
16	2310	22	68	22	22958	410	477
17	5251	36	80	23	38716	664	718
18	8155	60	125	24	59598	2429	2500
19	10285	97	180	25	109278	5289	5608
20	10836	155	238	26	185065	6555	6650
21	15128	252	299	27	290450	9918	10002

7 Conclusions

In our opinion, the value of the sum of several different distributed software environments can be increased by providing them with constructs enabling cooperation without the loss of the original computational model. The proposed *Interoperability Architecture* addresses this challenge. In this paper we have concentrated on adding well-known systems allowing declarative distributed programming, such as Strand and PCN, to IA. These systems are slow for general-purpose computing, but many applications, especially those demanding some decision taking and symbolic processing on one hand and heavy numerical computations on the other, may be written in IA profiting from multi-lingual (like object-oriented and logic programming) approach. We have tried to identify the common features of declarative systems which allow to generalize certain constructs and ideas when adding the system to IA. We may state that there exist some general guidelines along which the construction of stub, linking a declarative system and IA, can be created. We have presented them along with two ways in which a declarative system can handle requests incoming from IA: conversation and registration modes. The paper contains the description of successful implementation of the described mechanisms in Strand and PCN. It may be a strong evidence that our solutions and proposed methods are general and applicable.

References

[1] *ANSAware 4.0 — Application Programmer's Manual*, APM Ltd. Cambridge (1992).

[2] G. Attardi, M. Gsapari, *Multilanguage Interoperability*, TR UBLCS-93-18 Bologna (1994).

[3] *Draft Common Object Request Broker Architecture Revision 1.1*, OMG Report 91-12-1, (1991).

[4] I. Foster, S. Tuecke *Parallel Programming with PCN*, Argonne National Laboratories, (1992).

[5] I. Foster, R. Olson, S. Tuecke *Productive Parallel Programming: The PCN Approach*, Scientific Programming, Vol.1 (1992).

[6] I. Foster, S. Taylor, *Strand - New Concepts in Parallel Programming*, Prentice Hall (1990).

[7] R. Hayes, S. W. Manweiler, R. D. Schlichting, *A Simple System for Construction Distributed, Mixed–language Programs*, The University of Arizona TR 87–1b, (1987).

[8] F. Nayeri, B. Hurwitz, *Dispatching Experiments in a Distributed System*, GTE Lab. Inc. (1993).

[9] *Strand'88 User Manual Buckingham Release*, Strand Software Technologies Limited, (1990).

[10] A. S. Tanenebaum, *Modern Operating Systems*, Prentice-Hall (1992).

[11] A. Uszok, G. Czajkowski, K. Zieliński *Interoperability Gateway Construction for Distributed Object-Oriented Systems*, Proceedings of 6'th NWPER'94, Lund, TR Lund University (1994).

[12] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, P. Tarr, *Specification Level Interoperability*, Com. of ACM, may vol. 34 (1991).

[13] K. Zieliński, G. Czajkowski, A. Uszok, *Interoperability in Open Distributed Processing*, Technical Report, Institute of Computer Science UMM Cracow (1994).