Transparent Interoperability in Distributed Processing

A. Uszok, K. Zieliński, G. Czajkowski
Institute of Computer Science, University of Mining & Metallurgy
Kraków, Al.Mickiewicza 30, 30-059, Poland
e-mail: {uszok, kz}@ics.agh.edu.pl

Abstract

This paper[1] *proposes a framework architecture of the system enabling constructions of distributed, complex and heterogeneous applications. In order to make it possible a concept of transparent interoperability between different distributed systems was introduced. It is understood as unconstrained cooperation of different distributed environments, regardless of their computing models, grain of parallelism and most of other features. Methodology of application construction in the originated software heterogeneous environment concludes the paper.*

1 Introduction

Today, demand for more intelligent and cooperative information systems challenges their developers. It will surely result in a creation of larger and inherently more and more complex systems. The solution to this can be construction of systems as a composition of independent modules. It could be very desirable and sometimes even necessary to employ diverse programming languages and environments in implementations of such modules. In the realm of distributed processing, many systems already exist [5, 6, 8, 9, 10, 11] that offer various models of computation, suitable for different purposes. Thus an environment enabling integration of modules programmed in different computational models is very expected.

One of the first attempts at constructing the multi-paradigm environment was the Mixed–language System [1] from the University of Arizona and the Q system [3] from the Colorado University. Now, CORBA-complaint [5] systems are gaining wide attention. However, existing interoperability environments are aimed at the integration of processes, written in different programming languages, into a set of cooperating entities and for this purpose they define their own proprietary model of distributed processing.

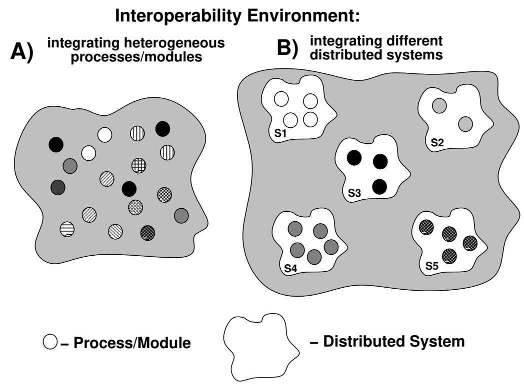

Figure 1: Interoperability between heterogeneous processes versus interoperability between different distributed systems

Figure 1A shows such case when environment puts together many heterogeneous processes/modules but organises their cooperation according to one chosen model. For instance CORBA-complaint systems impose an object-oriented model of computation and integration to which every aspect of cooperation is submitted. The new situation originates when the constructed environment is aimed at interoperability between entities which are already distributed inside, like different distributed systems, Fig. 1B. Such systems consist of a set of interacting homogeneous processes, but processes/modules from different systems cannot interact because of their inherent heterogeneity. On the first sight it seems that the situation is very similar to the previous one; interoperability environment has to solve problems of cooperation between heterogeneous modules. However, there is a large difference –

[1] This work was sponsored by the Polish State Committee of Scientific Research (KBN) under grant no.: 8T11C01210
Grzegorz Czajkowski is currently at Department of Computer Science, Cornell University, Ithaca, NY 14853, email: grzes@cs.cornell.edu

these entities already possess their own models of distributed computation, which could be chosen from a large set of possible variants: remote procedure call, client-server, asynchronous messages passing, CSP-like synchronous randez-vouz, shared dataspace, tuple space, data flow model etc. The environment integrating them has to preserve these existing computational models. Naturally, the interoperability environment introduces certain model of computation which organises cooperation. However, if this model is flexible and extend-able enough to transfer the computational models of interoperable entities between them and does not violate the paradigm of computation inside them, than it will be called **transparent interoperability**.

The problem encountered here can be compared, in some aspects, to the problem of interoperability between different CORBA-complaint systems [4]. They are created according to one standard, but differences in functional and technical solutions make them unable to interact directly. There are many distinct domains in these systems (security, management, visibility, lifetime, distance, etc.), which are results of various areas of their applications. These execution boundaries have to be transparently crossed by services invocations. Thus, some solutions employed in Inter-ORB interoperability [15] may be also applied, after necessary extension, to the transparent interoperability of distributed systems. However, as the matter of fact, computational models employed by CORBA-complaint systems are the same, so the problem with mapping one model of distributed computation to the other does not exist in this case.

In this paper, a framework, called Interoperability Architecture (IA) is proposed for achieving transparent interoperability of various distributed environments. It is assumed that this framework enables unconstrained cooperation of different environments, regardless of their computing model, grain of parallelism and most other features. Three basic components of the IA have been identified: IA Protocol, IA Server and Stub.

The rest of this paper is organised as follows. In the next section requirements of transparent interoperability are described in detail. In Section 3 IA framework components are presented and discussed. Variants of the framework specialisation are given in Section 4. Software design methodology in IA is elaborated in Section 5 and illustrated by an example. Finally, in Section 6 concluding remarks are made.

2 Determination of Requirements of Transparent Interoperability

An investigation of various distributing environments (ANSA [6], CORBA [5, 7], SR [8], PCN [10], Strand [9], Linda [11] etc.) representing very different models of distributed computation and technical solutions, as well as the existing interoperability schemes, leads to the designation of the following general features, which should be supported in order to fulfil the transparent interoperability requirement:

- **fine-grained integration**
  The fine-grain integration means that functions and services written in different systems may be used transparently. It is in contrast to the integration of whole applications or large modules, without use of their original models.

- **extend-able component model**
  It functionality has to be basic forming a framework for possible extension. It is supported by the "interface" component model as oppose to a "construction" model. What means that the model specifies way in which object interfaces are expressed without deep knowledge in their internal construction.

Defined above general transparency features have to be assisted by technical solutions included in the lower levels of the interoperability mechanisms. These levels concern: communication protocol that defines uniform data representation and an interaction scheme, environment interfaces that provide mechanisms of services, performed by systems integrated within the IA, invocations, and infrastructure for these invocations dispatching consisting of trading system and binding mechanisms. Chosen solutions on these levels could have different characteristics. However, in order to maintain transparency, these solutions have to possess the following features:

- support for asynchronous communication,

- extend-able protocol for various data types,

- language constructs to interoperability mechanisms fitted to the environments' languages semantics,

- support for different models of coordination and synchronisation,

- support for binding in heterogeneous environment (trading, repository for heterogeneous components, etc.).

Next section will present how the IA framework incorporates these features in its functionality achieving transparent interoperability between distributed systems.

3 Interoperability Architecture (IA) Framework

The IA architecture consisting of three building elements: IA Protocol, IA Server and Stub is presented in Fig. 2.

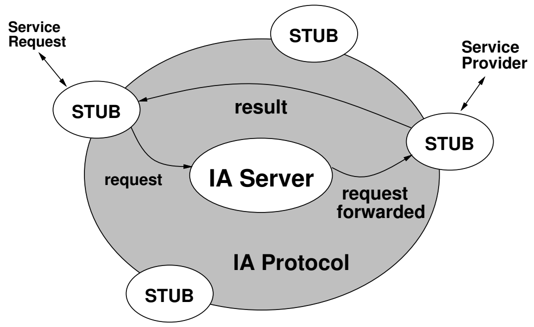

Figure 2: The general architecture of the IA Framework

The IA Protocol is the only component of the IA Framework which had to be fully specified and creates a backbone of the system. This protocol provides basic communication mechanisms, exception and error handling. The protocol is designed in a way supported transparent cooperation between processes of service client and service provider. In contrast to the IA Protocol, the IA Server and the Stub represent the subframeworks that may be refined or specialised in various way. They should preserve external interfaces and provide strictly defined functionality but their internal implementation is not fixed. The IA Server performs binding and invocations dispatching functions and may be implemented in centralised or distributed manner. The Stub is responsible for data type translation and service invocations and may be refined in a few well defined steps for any target distributed environment. This refinement process is described in more details in section 4.

In this section components of the IA framework architecture are described in general and significance of their construction for transparent interoperability achievement is evaluated.

3.1 IA Protocol

3.1.1 Base transport protocol

For us the only reasonable choice concerning data transmission protocol was TCP/IP versus UDP. Features favouring UDP were: preserving boundaries between messages and efficiency. On the other hand, while a message that is delivered using UDP is (almost) never distorted, messages do sometimes get lost or duplicated. Reliable use of UDP requires setting up an acknowledgement/time-out protocol. Although TCP does not possess this disadvantages, we found that for our purposes efficiency is one of the moscrucial feature and choose UDP. A considerable amount of time is anyway used by type converting and marshalling procedures, so at least communication protocol should be efficient. Also, LAN was our target environment, where cases of network errors and disconnection are not often.

3.1.2 Uniform data representation

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 Prolog? 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 (it is difficult to imagine a language that would not provide that): integers, reals, strings and sequences of these types. 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.

Protocol data types encoding is characterised by common form representation which imposes a big-endian byte ordering and explicit typing. The data types representation incorporates type-pattern concept which makes it extensible, enabling construction of the encoding for the new data types in the same uniform way.

3.1.3 Error and exception handling

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* 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 that supplied or it may return result of type different from the one caller expects (this errors may be common when programming using type-less languages with strongly typed languages). All of these errors are reported to the caller, using error reporting mechanism available in a given environment.

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.

A state diagram and message types implemented by the IA Protocol are determined by the brokering scheme described in section 3.2 and the presented error and exception handling mechanisms. They are elaborated by [16].

3.2 IA Server

Functionality of the IA Server concentrates in trading and brokering function. It maintains data base in which environments register offered by itself services. Using this information it routes requests which come from environments and are directed to remote services. The IA Server's internal architecture could be centralised (simpler) or distributed (more reliable, no bottleneck). The centralised version is implemented as one process, which serves as a *trader* and *broker* for all cooperating environments. The distributed version could use some existing distributed environments, like ANSA or CORBA-complaint system which can serve as a medium which localises and routes IA requests (see Fig. 3).

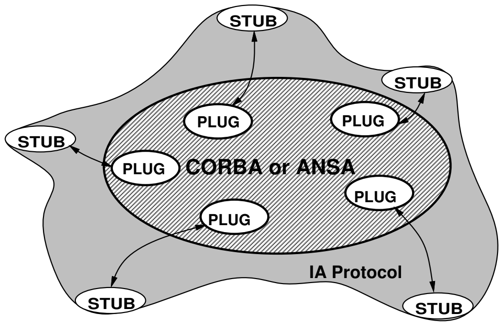

Figure 3: CORBA-complaint system (ANSA) as a *medium* in distributed architecture of the IA Server

The chosen environment for the IA Server implementation may be seen as a *glue* which keeps together elements of the IA. To achieve this we had to design component which will represent given system in the medium environment. We called this module *plug*, as it connects a system to the medium. It is a regular *glue* environment object. It receives remote operation invocations from the environment, translates them into *glue* system remote operation calls format, and forwards it to another *plug*, which represents the *server* environment. The medium environment is responsible for binding and dispatching of request calls to the *plug* of the system which offers the required service.

3.2.1 Requests brokering in the IA

Two nowday popular brokering architecture are presented in the upper part of the Fig. 4. The first variant uses the *Servers broker* only to localise the required service, then client contacts the provider of this service directly. The example of such system is ANSA with its Trader. The second variant always forwards requests through the some intermediary broker which routes requests to the appropriate service provider. The example of its

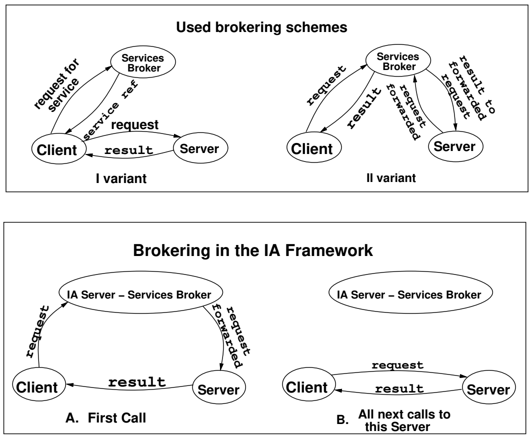

Figure 4: Brokering schemes

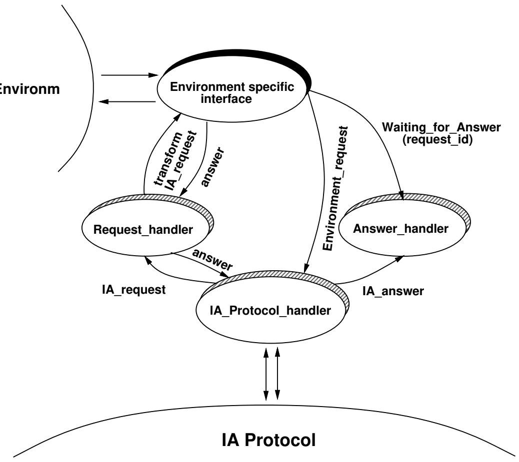

Figure 5: Stub Sub-Frameworks Architecture

usage could be most of CORBA standard implementations.

The IA Frameworks uses the new, not yet employed scheme of requests brokering. Client sends its request to the IA Server, which finds provider of the desired service and forwards the request to it. The IA Server informs the client where its request has been forwarded. The answer from the service provider is however sent directly to the client. Also, all next requests from this client to the same server will be sent directly to it. The IA brokering scheme reduces number of nodes in the path of the first request invocation from four to three.

3.3 Stub

The *Stub* is responsible for all activity related to environment cooperation with the external world. This part of the IA usually can be written in the language offered by the environment, actually at least some parts of it have to be written in this language. It results from the fact that this component of the IA should call foreign services in the most transparent manner for the environment for which works as well as put in the environment external calls in the same manner. The important aspects of Stub construction is its asynchronous behaviour. It has to be able to serve many calls to and from the environment simultaneously. The threads or event-driven mechanism has to be used for this purpose – it largely complicates Stub implementation.

In general Stub actions during invoking external service are:

- Choosing and take over invocations of external procedures from an environment,
- Translating them and their arguments into universal form,
- Attaching unique identifier to this invocation,
- Putting record with this identifier into the list of calls waiting for an answer,
- Sending such prepared form of invocation to the binding service,
- Receiving answers and errors from others environments.

Fig. 5 shows universal structure and elements of *Stub* Sub-Framework. Particular components perform the following operations: *IA_Protocol_handler* receives massages from and to the environment, it also maintain *Keep-alive* and *Acknowledgement* sub-protocols. *Request_handler* serves requests from the IA, usually it also maintains data base of services offered by the given Environment Stub to the external world. *Answer_handler* matches answer which came from the IA with the environment calls which wait for them. *Environment specific interface* has to be constructed for the given environment as it takes over invocations of remote services from the system as well as puts in it requests from the IA.

4 Stub Sub-Framework refinement

This section presents specialisation of the Stub Sub-Framework for two different families of distributed systems. Refined parts of the Stub have black border in the figures.

4.1 Stub for declarative environments

The Stub (Fig. 6) for this class of systems [12] is characterised by the *stream-like* interface from the environment side, two layer implementation: upper layer written in the environment's language and the lower layer written usually in C. The architecture incorporates two ways in which requests, incoming from the IA, could be handled: conversation and registration modes. The construction of this Stub is not service specific. It could be used, when build, for handling requests for any service from both sides: the IA and the environment. This is big advantage of this design, which however was feasible because of declarativness of these systems. Successful implementation of the described mechanisms in Strand and PCN has been achieved.

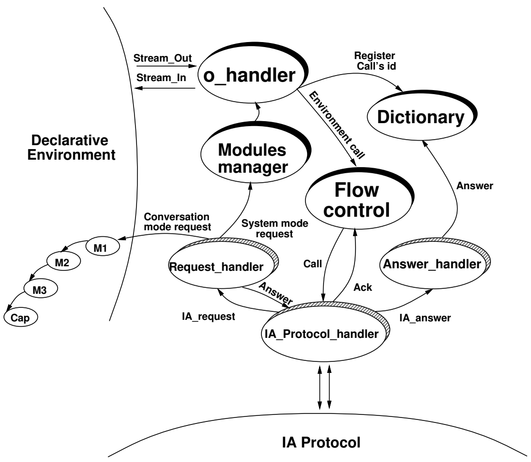

Figure 6: Stub refined architectures for declarative systems

4.2 Stub for object-oriented environment

The Stub in this case (Fig. 7 is an independent object which exists in the system [13]. Its task is forwarding and translation of messages between the environment and the IA Protocol domain. It was called a gateway, as it opens the environment to the outside world. The impossibility of generic construction of this module for some environment (ANSA, SR) has been identified. In this case the interface specific versions have to be created for each exported or imported service. However, gateway architecture has been divided into to parts: universal and service dependent. This latter (*Client_Stub* and *Server_Stub*) may be created automatically, thanks to their structure, using specification of services interfaces. This has been proved by creation of preprocessor for ANSA systems to perform this task. Some object-oriented systems (CORBA-complaint) enable construction of a generic gateway. In this case sophisticated mechanisms like Dynamic Invocation Interface and Dynamic Skeleton Invocation [5, 4, 15] have to be used.

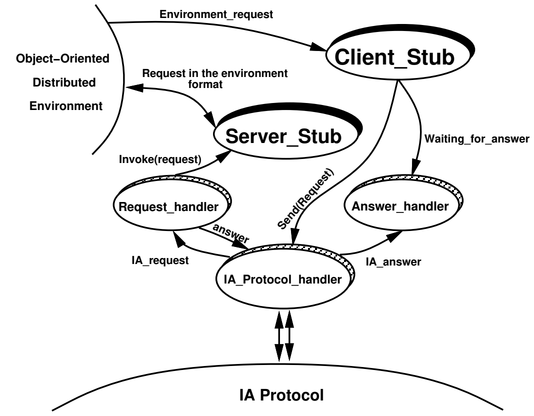

Figure 7: Stub refined architectures for object-oriented systems

5 Software Design Methodology in Interoperability Architecture

While designing an application which is to be completely implemented in a single language (the situation most common today), one must necessarily take into account the limitations imposed by the chosen language. For example, it is much easier to use lists in functional languages rather than try to construct indexed data structures, simulating arrays. Similarly, lack of destructive assignment in most of the logic programming languages may make one think twice before deciding to use big data structures with small update-able components. On the other hand, different languages have their advantages – for example higher order functions in functional programming or *layered streams technique* in logic programming which allows optimised storage of partially identical results. To cut the long story short, the features of the language influence the design process.

There is a better approach, base on assembling a program using optimised and debugged procedures from existing libraries. This is, however, the *black-box* approach to software development process. Procedures are treated as objects with a known interface but one can make no assumptions as to their implementation. This, although of great practical importance, will not be of our concern because we are more interested in writing programs here, not only in putting them together.

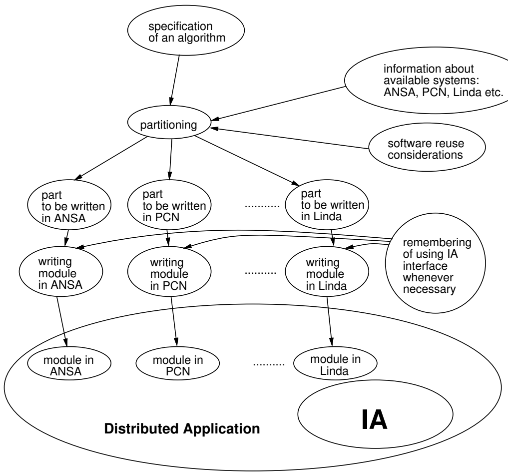

Figure 8: Software design methodology in IA

In a multilingual approach, exhibited in the IA, a software design methodology different from the two mentioned above is needed. We would summarise it in the following general steps, taking as an input the description of an algorithm (Fig. 8) to be implemented:

1. partition the algorithm into the set of modules[2], P_1 .. P_N, which are responsible for different actions (eg. decision making, pattern matching, numerical computations, database services); use the knowledge concerning modules already written,

2. take into account the information about languages available in IA and assign the modules from step 1 to the most appropriate languages; let the language assigned to P_i be denoted as $lang(P_i)$,

3. implement modules in the assigned languages, remembering to call the module P_i from module P_j using interface to IA if $lang(P_i) \neq lang(P_j)$,

4. if any module P_i may be called by any other module P_j and $lang(P_i) \neq lang(P_j)$, remember of registering P_i in IA.

[2]The name module is used here to mean a component which has a well-defined functionality nad interfaces

The example of usage the IA and presented above methodology will concern computing a colour bitmap of the Mandelbrot fractal. The *farmer-worker* algorithm has been chosen for computation organisation. In this algorithm the *farmer* is responsible for the management of the computation which is carried on by *workers*. So the *farmer* task is more decision oriented while the *workers* is calculation oriented. From the available systems it has seemed that the ANSA systems, which provides C distributed objects, were the most appropriate for *worker* implementation. The declarative system Strand has been chosen as a environment for "intelligent" *farmer* (Fig. 9). The overhead of usage the IA when calling a remote operation was average 88ms (Sun workstation 2). This simple example shows how the IA could be used for development of much more sophisticated applications. The IA framework was also used during construction of the Interoperability Layer (IL) for the MODIMOS monitoring system [14]. This system enables monitoring of activity in different distributed environments at the same time thus it needs a mechanism (IL) for collecting data from all these environments and managing locals monitors working in them.

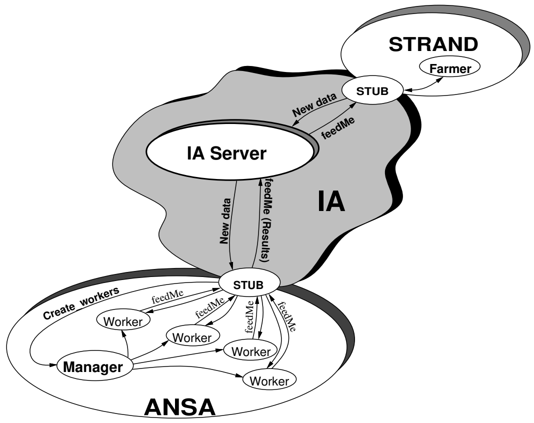

Figure 9: Software heterogeneous application in the IA – computation of fractal's bitmap image

6 Conclusions

The presented framework for the Interoperability Architecture specifies how an uniform platform for hybrid distributed processing could be build in the structural way. The methodology of construction of large and complex application in such a hybrid and heterogeneous environment was as well described. Various computational and coordination models may cooperate effectively within one application, being implemented in the most suitable system, in the environment constructed using this framework. Framework's architecture and functionality have been presented with the stress on these its features which support transparent interoperability. The only element of the framework which has been fully defined is the IA Protocol. The design of this protocol determines the functionality of sub-frameworks and creates a backbone of cooperation.

The IA Server and the Stub which are sub-frameworks have to be further specialised for the concrete requirements. Architecture variants of IA Server has been enumerated as well as implemented. Its distributed internal architecture uses CORBA-complaint systems (or ANSA as) a glue. The special module – plug has been introduced in this case which represents the Stub in the chosen glue environment. The most difficult task is construction of the Stub for the given environment. Thus its general architecture has been presented together with the specialisation process of the Stub Sub-Framework for declarative and object-oriented systems. Stub modules for these class of environments were implemented and tested.

The prototype environment created using this framework integrates ANSA, SR and Orbix object-oriented distributed systems with Strand and PCN declarative distributed systems and Linda, which offers tuple space model of distributed coordination. This methodology of achieving interoperability of distributed systems was also used in other research architectures, as for instance in a monitoring system – MODIMOS [14].

References

[1] 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).

[2] B. Janssen, D. Severson, M. Spreitzer, *ILU Reference Manual. Version 1.6.4*, Xerox Corporation (1994)

[3] M. Maybee, D. Heinbigner, L. Osterweil, *Multi-language Interoperability in Distributed Systems: Experience Report*, TR of Universityy of Colorado (1995)

[4] *CORBA 2.0/ Interoperability. Universal Networked Objects*, OMG Inc., TC Document 95.3.10 (1995)

[5] *CORBA 1.2 Revision Draft*, OMG Report 93-12-43, Object Management Group (OMG) Inc. (1993)

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

[7] *The Orbix Architecture.* IONA Technologies Ltd., (1993).

[8] G. Andrews, et al., *An overview of the SR language and implementation*, ACM Transaction on Programming Languages and Systems, 10(1):51–86 (1988)

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

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

[11] N. Carriero, D. Gelertner, *How to write parallel programs: A guide to the perplexed*, ACM Computer Surveys, September (1989).

[12] G. Czajkowski, A. Uszok, K. Zieliński, *Distributed Declarative Systems as Parts of Cooperating Software Environments*, Proceedings of ICLP'94 Post-Conference Workshop 'Integration of Declarative Paradigms', Santa Margerita, Italy (1994)

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

[14] K.Zieliński, A. Laurentowski, J. Szymaszek, A. Uszok, *A Tool for Monitoring Heterogeneous Distributed Object Applications*, Procceedings of ICDCS-15, Vancouver, Canada, IEEE Computer Society Press (1995)

[15] M. Steinder, A. Uszok, K. Zielinski, *A Framework for Inter-ORB Request Level Bridge Construction*, Proceedings of IFIP/IEEE ICDCP'96, Drezden, Germany, Chapman & Hall (1996)

[16] A. Uszok, *Transparent Interoperability in Distributed Systems*, PhD Thesis, University of Mininig and Metallurgy in Cracow (1995)