A Coordination Kernel for Coupling Heterogeneous Programming Environments *

Frédéric Bapst Oliver Krone
Institut d'Informatique, Université de Fribourg
Fribourg, Switzerland
Frederic.Bapst@unifr.ch, Oliver.Krone@unifr.ch

Abstract

The aim of the present paper is to propose a model to coordinate agents running in heterogeneous programming environments. During the design of an application, a single programming language is not always sufficient to express the different subproblems in an elegant way. One solution is multi-language programming, which raises the problem of coupling different programming environments. Our goal is to define, for such distributed applications, a uniform coordinating architecture, general enough to couple languages relying on heterogeneous paradigms (imperative, object-oriented, declarative, constraint-based, concurrent, event-based etc.). First we describe a way to apply a universal agent notation in various language constructs, then we propose our message passing based coupling scheme: both a precise specification of new concepts and their implementation are discussed. We show how a twofold interface definition language would improve the development phase itself. On top of our architecture, a simple programming methodology is presented. Concrete examples and applications are sketched, using among others PT-PVM, Tcl/Tk and Oz.

Keywords: Coordination, software architecture, multi-language programming, heterogeneity, message passing, interoperability, software reuse, distributed applications, interface definition language.

1 Introduction

The integration of different programming environments becomes more and more crucial with the specialization of the environments, and with the growing size of the applications to realize. Some algorithms are easier expressed in a particular specialized environment and it would be artificial to implement them in another. For instance, user interface development seems more cumbersome and error-prone in C-Xlib [21] (at least for newbies) than in a scripting language like Tcl/Tk [23]. In general, real-size applications involve a vast scope of computation skills, and it is tempting to use the best programming environment for each subpart. To make the coupling easier, one can aim at either case-tailored remedies (see, e.g., Oz [13] interfacing Tcl), or a general solution (which is our ambition).

Our objective is to set up a coherent framework to improve multi-language programming, helping the programmer to focus his attention on the application, and not on internal details. Basically, the idea is to facilitate in a uniform way the interactions across multiple programming environments with message passing as the basic underlying interaction protocol. Here are some typical practical problems that we address:

*This paper was submitted to the Second International Conference on Coordination Models and Languages (COORDINATION'97, September 1-3, 1997, Berlin, Germany).

- *Compiled/interpreted*: suppose you implemented an application kernel in C using Unix threads. Somebody else designed the front-end with an interface builder that generates Tcl-Tk code. How can you redirect the Tcl-Tk callbacks to the threads running in the C environment?

- *Declarative/imperative*: there is a very powerful constraint programming tool running in Oz [13], and you want to reuse it within an Ada simulation program. How can you mix these very different programming paradigms?

- *Logic parallel/object oriented*: you want to design a tracing facility to show the evolution of a Strand [10] program. Your idea is to associate a Java [9] object to every Strand lightweight process running in the pool. What is the difficulty of encoding this coupling?

To the best of our knowledge, this problematic has not yet been studied intensively. However, at least two prominent developments, ToolBus [5] and Corba [1] have already addressed this problem. In the following we will discuss their point of view and compare it to our approach.

ToolBus discusses the coupling of software tools that are available in executable form. It helps writing applications that take advantage of existing utilities (e.g. `gnuplot`). Our basic motivation is a bit different, because we are mainly concerned with applications where some packages would be better developed in a certain programming environment. In other words, we take the problem earlier, that is from the management of the sources. We agree with the authors of [5] that the following two important aspects for heterogeneous software coupling have to be taken into account:

- *Data integration*: how can data be exchanged and shared between different environments; proposed solutions include the ASN-1 standard [12] and the XDR [19] format.

- *Control integration*: how can processes communicate with each other, what kind of communication paradigm should be used, e.g., coupled communication or uncoupled communication [7]. We chose the message passing paradigm.

Corba shares with our proposition the goal of providing a common programming environment for distributed applications across hardware and software platforms. Even though the comparison seems a little bit daring (because of the much greater complexity of Corba), we nevertheless state some of the differences and similarities:

- Corba achieves its platform independence by imposing the full object oriented programming model. We have chosen a different approach by restricting ourself to the exchange of messages. We are convinced that this least common denominator for communication pattern on different hard/software platforms facilitates the integration of a wide range of programming environments.

- The functionality of our agents is closely related to the environment where it lives, whereas in Corba object functionality can be realized in different programming languages (typically for commercial systems: C++, or SmallTalk). The strong relationship between our agent functionality and its "home" environment is forced; we therefore plan to exploit the particular strengths of each programming environment, by means of an "integration glue".

- Both mechanisms use a platform-independent specification language for the definition of the (static) interface for their distributed application. Our approach is twofold: we use an Agents Definition Language (ADL) and a Countries Definition Language (CDL). Corba instead, uses an Interface Definition Language (IDL) for object specification.

- Corba's Remote Method Invocation is closely related to synchronous client/server programming, even though Corba allows to invoke methods asynchronously. Our model however, uses per default an asynchronous message passing model as its interaction model. Our approach can therefore be classified as a Message Oriented Middleware (MOM) [22] approach.

The remainder of this paper is organized as follows: in Section 2 we introduce a unifying metaphor for all programming models, which motivates the generic aspect of the problematic. Section 3 is devoted to the description of the proposed mechanism from theoretical issues to realization with an emphasis on the specification. A new programming methodology is presented in Section 4, to show both the expressive power of our approach and the practical advantages for the programmer. Finally, some conclusions are exposed.

2 Programming Environments as Multi-Agent Countries

2.1 Looking for a Common Kernel for all Languages

Before proposing a coupling architecture, we specify in this Section *what* is to be mixed. From the programmer's point of view, each programming environment represents above all a computing model which comes with precise constructions and requires a certain flavor of know-how. There are generally two ways to cope with different programming environments and hence with multi-language programming:

- Define a construction in one language (e.g. declaring a Modula-2 procedure tagged with EXTERNAL) and implement it in another (e.g. a C function body). The bridge will be set up at link time, or through the generation of a new kernel (e.g. an enriched Tcl interpreter). But this approach is clearly not generic, because at least one of the two environments must have been explicitly designed to accept the other, which is a great limitation.

- Build up several programs that will be joined at runtime by I/O (e.g. a Unix pipe), which is a very general scheme. Then, with a focus on process execution, a programming environment represents a *coordination space* (see Section 3.1), in the sense that all activities happening at runtime have a natural "scope", isolated in a dedicated runtime context. It is on that coordination space paradigm that our coupling method is based.

To allow a coherent coupling scheme, we need to argue that some concepts appear, under a certain form, in every programming environment. What we introduce is a slightly extended metaphor, where each coordination space is as well considered as a *multi-agent country*, i.e. a world with its own laws, occupied by living entities. The next Section addresses the specification of these active components. We will see in Section 2.3 that this is a quite reasonable common denominator.

2.2 Definition of a General Agent Protocol

The only hypothesis on the programming environments to be coupled is that they all succeed in expressing the notion of agent. Here, an agent is an abstract representation of active entities, having a certain behavior and interacting with one another. As summarized in Figure 1, we impose the following protocol to be respected:

- *Naming protocol* – An agent is considered as one instantiation of a generic template. There is a way to designate both an agent kind and a specific agent.

Naming protocol	`TYPE Agent=...; TYPE AgentKind=...;`
Spawning protocol	`PROC Spawn(IN AgentKind k, OUT Agent new)`

Interaction protocol	`TYPE MsgKind=...; TYPE MsgArgs=...;`
	`PROC Post(IN Agent to, IN MsgKind m, IN MsgArgs a)`
	`PROC Handle(OUT MsgKind m, OUT MsgArgs a)`

Figure 1: Our agent protocol.

- *Spawning protocol* – Agents are generated dynamically; this operation requires an agent kind, and will return the name of the new born.
- *Interaction protocol* – The manifestation of the agent's influence on the environment and conversely is conveyed by interaction requests. There is a means for the agent to post them, and another to receive (handle) them. Moreover, an interaction request is tagged with a semantic label denoting the kind of event, and may vehicle extra information under a certain form.

Not every language was designed to respect this specification in their built-in constructions, but we pretend that they all offer a way to express these concepts. So the agent notion has an analogous sense in different countries, whatever local laws fixing their rights and obligations.

2.3 Language-Dependent Expression

The agent specification discussed above may seem a bit ambitious to be universal, but in fact the programming environment must merely provide a facility to isolate the execution of each agent (to preserve its identity), and a means to activate them (via an adapted execution engine). This can be achieved through heterogeneous support:

- *Encapsulation mechanism* – The archetype is the object concept, because it clearly defines the state and behavior through attributes and methods. But we can also use an abstract data type or a package, where each routine call on an instance reflects a piece of behavior. It may be a single procedure with an internal switch over caught events. For a declarative language, it would be a group of logical rules. The agent may also be a binary program coming with a command line interface. The most prominent representative for this kind of approach is the Corba [1] object oriented framework.
- *Activation mechanism* – A direct implementation is through concurrent/distributed processes and message passing like in PVM [11] or, with an emphasis on generic message passing, in SUN's ToolTalk [15]. An intrinsic event handling like Tcl-Tk will also be sufficient. Finally, we can simulate an application main loop (e.g., waiting on input data), and activate agents through a routine call on a particular variable carrying their internal state.

Figure 2 shows how the agent protocol is expressed in 3 languages of different nature: an imperative one with message passing facilities (C with PT-PVM [17]), an interpreted scripting one with object-oriented extensions (iTcl [23]), and a concurrent logic one (Strand [10]). In these examples, the agent concept is very natural. In general, this view is not surprising, because programs tend to be structured, and our proposal involves a structural tuning.

The wide applicability of the agent abstraction is a strong requirement, because the real advantage of setting bridges between languages is to combine the strengths of various computing models, or to reuse solid existing pieces of code. We claim that our architecture will succeed in coupling

Pt-pvm	iTcl	Strand

`void page_thread(...)`
`****** INIT BEHAVIOR ******`
`  ...`
 	
`do`
`********* HANDLE **********`
`csReceive(...);`
`switch(msg_tag) {`
`case addBlockMsg:`
`  ...`
`********** SPAWN **********`
`  csSpawn(bk_thread,&new);`
`********** POST ***********`
`  csSend(new,flashMsg,&buf);`
`...`
`} while(1); }`
	`itcl_class Page { ...`
	`****** INIT BEHAVIOR ******`
	`method constructor{...} {`
	`  ...`
	`}`
	`********* HANDLE **********`
	`method add_block {x y} {`
	`  ...`
	
	`********** SPAWN **********`
	`  set new [Block #auto]`
	`********** POST ***********`
	`  $new flash 4`
	`}`
	`}`
		`start_page(Is,...) :-`
		`****** INIT BEHAVIOR ******`
		`... ,`
		`page(Ms,...).`
		
		`********* HANDLE **********`
		`page([addBk(X,Y)|Is]...):-`
		`... ,`
		
		`********** SPAWN **********`
		`start_block(NewStream,...),`
		`********** POST ***********`
		`NewStream:=[flash(4)|NS1],`
		`...`

Figure 2: Our agent protocol expressed in 3 different languages.

heterogeneous components, such as Oz [13] and Java [9] objects, Ada [3] tasks, Unix RPC servers [6], X11 [21] widgets, Coala [4] agents, Atome [18] experts, Labview flow-based modules [14], or Khepera [20] robots.

3 Description of the Proposed Mechanism

The goal of our proposal is to extend the agent protocol described in the previous Section to allow the coordination of agents running in different *countries* (programming environments). We want to design a set of generic facilities for remote agent creation (creation of agents executing in a different country) and communication. The fundamental idea is to offer dedicated commands to carry out the `Spawn` and `Post` operations towards a foreign country, and to enable remote interaction requests with the natural `Handle` construction of each environment. Here are some motivations for this policy:

- The way to program within a language should not be disturbed by these new external facilities; it is utopian to override the basic constructions;
- Our agent protocol is a least common denominator; it does not prevent a programming model to offer some richer facilities (e.g. filtering on message reception or broadcasting);
- `Post` and `Handle` are intrinsically asymmetric. The former happens during a determined execution context; it is the manifestation of a *voluntary* behavior. The latter is not always supposed to respect a unique expected schema; it is often an event whose content will *wake* the agent and condition its reaction.

3.1 General Principles – Coordination Spaces

What we will describe is a precise specification of the theoretical framework introduced in [16]. Let's first summarize some formal notions of this problematic.

Programming heterogeneous environments is concerned with the *Coordination* of (1) concurrently operating entities running in a specific environment (intra coordination), and (2) the coordination between different environments (inter coordination). In the following we will focus on inter coordination of different heterogeneous programming environments.

According to the model introduced in [16], we have to rely on the notion of *Coordination Space* ($\mathcal{C}$-Space for short) as a representation of the programming environment. $\mathcal{C}$-Spaces are first

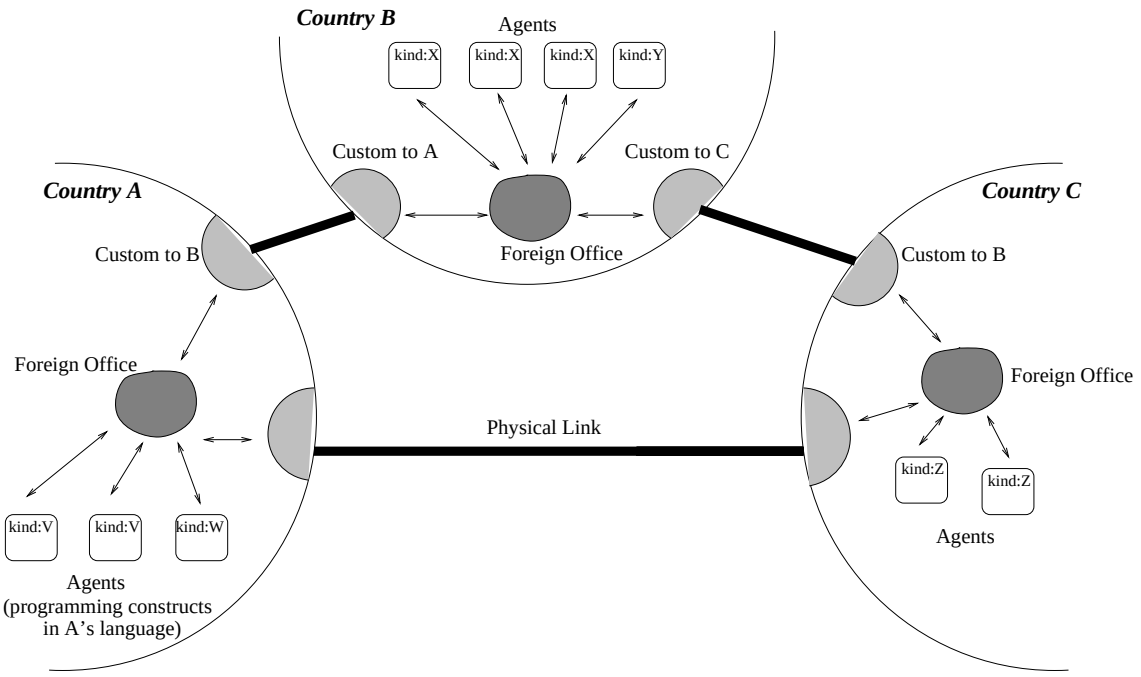


Figure 3: General coupling principles.

class objects; they have a global unique name and their identification can be transferred to other processes to access a formerly unknown $\mathcal{C}$-Space. Each $\mathcal{C}$-Space is defined by a triple: $(\mathcal{P}, \mathcal{D}, \mathcal{I})$, with $\mathcal{P}$ denoting the population of active entities running in this space, $\mathcal{D}$ a multi set of data items stored in this space, and $\mathcal{I}$ a control instance serving as the interfacing module between the different $\mathcal{C}$-Spaces.

Per default processes are visible within the $\mathcal{C}$-Space where they have been created and known by the process which created them (the father). The visibility can be extended by (1) putting the abstract process identification into the $\mathcal{C}$-Space, so that other processes can retrieve the information out of the $\mathcal{C}$-Space, or by (2) sending the identification to a process directly. The same is true at $\mathcal{C}$-Space level, that is their identification can be transferred to other processes too.

3.2 A Realization – Foreign Offices

The proposed realization of the theoretical framework introduced in the previous Section is the foreign office programming model.

A *Foreign Office* implements the control instance $\mathcal{I}$ of a $\mathcal{C}$-Space and serves as the interface between entities located in different $\mathcal{C}$-Spaces. To complete the metaphor, $\mathcal{C}$-Spaces are known in this realization as *Countries*, and processes (the population $\mathcal{P}$) wishing to interact with other ones running in a different country have to use a *Visa*. A visa is a global type that defines a unique representation of a correspondent and is used for the interaction with the outside world. Note that our realization does not include $\mathcal{D}$ of the $\mathcal{C}$-Space model. This is because it turned out that the implementation of such a multi set of arbitrary data items for several programming environments is (1) too complex for a generic case, and more importantly (2) not needed for the moment, because our approach is based on classical message passing. Therefore the model sketched in the previous Section is more general than our implementation.

Inside a country, the foreign office is the central instance which enables communication and

Declaration	Description
`TYPE AgentKind,Agent`	Agent template+samples (country-dependent)
`TYPE Visa`	String-encoded global IDs for the two types above
`TYPE MsgKind,MsgArgs`	Message passing format (country-dependent)
`CONST String here`	Name of the local country

`PROC ExportAgentKind`
`(IN AgentKind k,`
`IN Visa name)`
	Export an agent class to the outside world
	local template to export
	string respecting "here#here#fantasyName"

`PROC SpawnAbroad`
`(IN Visa kind,`
`OUT Visa new)`
	Create and start agent in other country
	template in the remote country
	the delivered visa for the new born

`PROC SendAbroad`
`(IN Visa to,`
`IN MsgKind msg`
`IN MsgArgs args)`
	Send message to a remote agent
	destinee
	msg tag (will be translated)
	related data (will be translated)

`PROC VisaFromAgent` `(IN Agent a, OUT Visa v)`	Deliver a visa if not yet registered
`PROC AgentFromVisa` `(IN Visa v, OUT Agent a)`	Get local correspondent identifier
`PROC CountryFromVisa` `(IN Visa v, OUT String c)`	Get country name where the visa owner lives

Figure 4: Foreign office specification.

hence coordination towards other countries, firstly by taking part in the distributed visa management, and secondly by mastering the inter-countries connection ports where all data conversions (which are necessary if processes communicate in different programming environments) will be held.

Figure 3 shows the general coupling mechanism between three countries. In each country an entity that wants to interact with another process in a different country asks the local foreign office for a visa, which will represent its global and unique identifier.

3.3 Specification

This Section describes the different primitives summarizing the services offered by the foreign office in order to interact with the outside world. The set of these primitives is structured in three parts: (1) types that allow to name an entity and its kind: (2) primitives that handle remote creation of new entities and (3) primitives allowing to interact between entities belonging to different countries. Figure 4 gives an overview of the foreign office specification, presented in a module-like fashion for convenience.

`AgentKind`: the way to name an agent template locally; in C with Unix threads, it would be a pointer to a C function.

`Agent`: the way to name a specific agent locally; in C with Unix threads, it would be the Unix thread identifier.

`Visa`: a string respecting the pattern "$\alpha\#\beta\#\mu$" where α and β are country names (see Section 3.5).

`MsgKind, MsgArgs`: the way to perform local message passing, for example strings (with spaces as separators for arguments).

`ExportAgentKind`: informs the foreign office that a local kind of agent will be accessed from the rest of the world under a certain name; this identifier is supplied by the user and has to respect the conventions defined for the visa data type (`Visa`).

`SpawnAbroad`: creates and starts an agent in a foreign country and returns the visa for this newly created agent.

`SendAbroad`: posts an interaction request towards a remote agent; this is a variation of the classical send primitive known for example from PVM.

`VisaFromAgent`: retrieves the visa for a locally-known agent, e.g., to communicate it to a remote agent. A new visa is delivered if that agent did not have one yet.

`AgentFromVisa`: inverse function of `VisaFromAgent`, returns the agent to a given visa. The precondition is that the visa represents an agent running in the same country.

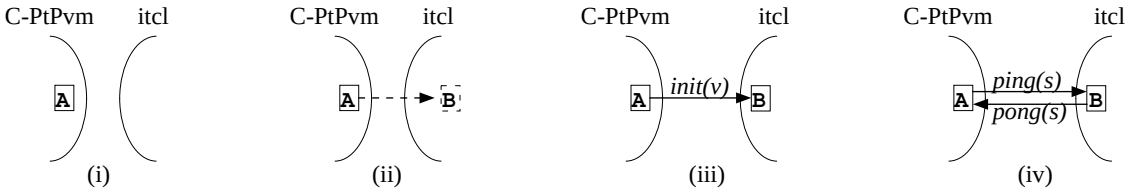
`CountryFromVisa`: returns the string encoded country where the corresponding agent resides. Thus an agent can detect if a certain visa represents an agent in the same country; in this case, they can then bypass the foreign office to interact directly using all the traditional primitives of that programming environment.

3.4 A First Program

Figures 5 and 6 present a small ping-pong program using our coupling architecture. One agent is written in C with the PT-PVM platform. The other is realized as an iTcl object. This toy example shows that bridging the language gap leads to a reasonable level of expressiveness, as can be seen looking at the implementation of each agent. The initialization of both foreign offices is performed in the main C program and the main iTcl script that are not shown here. The main C function must start the iTcl interpreter (`exec("Wish -f main.itcl",...)`) to get a pipe connection, and finally spawns the ping thread.

This example also points out another aspect of multi-language programming, namely the management of dependencies between different families of code. For instance, there is a constraint to ensure that `handle_ping` message tag understood by PT-PVM will be mapped to the string `"handle_ping"` on iTcl side. The use of literal expressions all over the program is certainly not the ideal solution because the programming environments (compiler or interpreter) will not be able to check these constraints. So the programmer is encouraged to choose constants/variable identifiers in a very disciplined way. The next step is naturally the use of an Interface Definitions Language (IDL), in order to automatically generate the carcasses of the multi-language software.

Figure 7 sketches how we designed such IDLs for our coupling mechanism, using the ping-pong program. This proposition splits the information into two parts:



(i) Startup: countries are set up and A is spawned. (ii) A (ping) spawns B (pong).
(iii) A sends his visa to B. (iv) A sends messages and B replies.

Figure 5: Ping-pong program, scenario.

Excerpt of C-PtPvm part (`ping.c`)

```
#include "pingpong_world.h"
    /* This header defines, among others:
         const char    AKPong[] = "itcl#itcl#AKPong";
         const MSG_TAG init = ...
         const MSG_TAG handle_ping = ...
         const MSG_TAG handle_pong = ...
         void  AKPing(THREAD_ENV *);
       and the foreign office operations.
    */
void AKPing (THREAD_ENV *p) {
 char his_visa[], my_visa[];  int i;

 SpawnAbroad(AKPong, &his_visa);
 VisaFromAgent(p->self, &my_visa);
 SendAbroad(his_visa, init, my_visa);
 for(i=0;i<10;i++) {
   SendAbroad(his_visa, handle_ping, "Hello");
   csReceive(..., handle_pong, ...);
 }
}
```

Excerpt of itcl part (`pong.itcl`)

```
itcl_class AKPong {
    #   method init        {arg1} {}
    #   method handle_pong {arg1} {}

  inherit pingpong_world
  inherit itcl_foreign_office
    # These classes define, among others:
    #   common handle_pong = "handle_pong"
    # and the foreign office operations.

  protected ping_visa

  method init {v} {
    set ping_visa $v
  }

  method handle_ping {msg} {
    SendAbroad $ping_visa $handle_pong $msg
  }
}
```

Figure 6: Ping-pong program, source code.

- The Agents Definition Language (ADL) specifies the different agent kinds that will be accessed across the frontiers. More specifically, it defines an identifier designating the agent kind, and describes the interaction requests that it understands, in terms of a message label and the types of the message arguments.
- The Countries Definition Language (CDL) specifies which programming environments will take part in the application and how they are connected, as well as which kinds of agents they will host.

The final syntax is still under development and will be extended. In Figure 7 uppercase letters denote a keyword, lowercase letters stand for a user-provided identifier, and words in square brackets denote capabilities (data types, connection means) accepted by both environments. The idea is then to provide a code generator that will take both ADL and CDL as input and produce the corresponding "templates" in each languages. In the ping-pong example, the header `pingpong_world.h`, the script for iTcl class `pingpong_world`, and the carcass of `pong.itcl` are expected to be output from the translation of the specification of Figure 7.

3.5 Implementation

As far as the implementation of the introduced concepts is concerned, we use the following methodology:

Physical connection – We require that each programming environment which will be integrated using this methodology, supplies one of the basic UNIX interprocess communication facilities such as pipes, sockets, shared memory, messages, signals or simply stdin/stdout.

Customs – Both ends of a physical inter-country connection is managed by what we call the Customs. This component is responsible to interface the foreign office with one of its international channel. This means parsing the input stream and feeding the output with two internal requests `remote_spawn(IN Visa kind, OUT agent)` and

Agents Definition Language (`pingpong.adl`)	Countries Definition Language (`pingpong.cdl`)

`AGENT KIND Ping`
`  HANDLES init  <visa>`
`  HANDLES handle_pong <string>`
 	
`AGENT KIND Pong`
`  HANDLES handle_ping <string>`
	`COUNTRY my_app  IS NEW <C-PtPvm>`
	`  REALIZES AGENT KIND Pong`
	
	`COUNTRY my_gui  IS NEW <itcl>`
	`  REALIZES AGENT KIND Ping`
	
	`CONNECTION my_app my_gui IS <unix_pipe>`
	
	`START my_app`
	`START my_gui`

Figure 7: ADL and CDL.

`remote_send(IN Visa to, IN MsgKind k, IN MsgArgs a)`. Finally, it has to translate interchanged messages content through encoding/decoding. Message translation becomes important when the agents are expected to exchange complex data structures.

Visa – The visa is a string encoded triple $\alpha\#\beta\#\mu$ where α is the name of the country where the owner executes, β the name of the country where the foreign office that delivered the visa is located, and μ an identification number. A foreign office that must deliver a visa knows the country name of the owner, and gives a fresh μ using a local counter. Thus μ is unique for this foreign office, $\beta\#\mu$ is unique globally, and α serves to situate the owner in its country, e.g., to root a message through the physical connections.

Distributed foreign registration – The association of every visa $\alpha\#\beta\#\mu$ and its owner is stored in a table maintained by the foreign office of country α. We have the guarantee that every "exported" agent or agent kind is registered somewhere, though there is no global name server. Besides, we can state that the implementation of our foreign office package requires at most one inter-country transaction per primitive, which is almost a mandatory constraint: handshakes on physical connections must be avoided because they would hinder the asynchronous management of communication streams.

Dynamism – The dynamic creation of communication links is an important property of our foreign office specification. The dynamic creation of new links between agents at runtime is possible because knowing a visa and any foreign office is sufficient to post messages, and also because a visa is a mere string that can be transmitted between agents (and over countries). Moreover, it is yet sufficient for a lot of applications to set up the multi-country world at compile time, but nothing prevents to add a country at runtime. We won't develop this topic here, because the foreign office initialization protocol is influenced by the connection means (e.g., a country connected via stdin/stdout has to be launched by another, whereas socket connection allow concurrent initialization). Note that the dynamic aspects also concerns the management of agent kinds considered in our proposition as high level constructions; it becomes interesting if the programming environment accepts the creation of new agent kinds at runtime, like Oz classes.

3.6 Examples of Foreign Offices

Now we can sketch the realization of the foreign office in different programming environments, with a focus on the semantic interpretation for the host language. Three case studies are addressed to

show some particularities in imperative (PT-PVM [17]), object-oriented script-like (iTcl [23]), and concurrent logic (Oz [13]) languages.

Pt-pvm – The home-made PT-PVM [17] is a distributed threads environment built on top of PVM [25]. The platform already supports the spawning of piped or socket linked Unix processes. In this programming environment, the foreign office is best expressed as a system correspondent, known by every (possibly distributed) thread that uses the international features. It is interesting to note that the multi-country paradigm can be used to extend the power of the environment, but also to revise some implementation choices. In fact, the realization of PT-PVM raises non-trivial problems like connecting threads running in different heavyweight processes, or accessing C functions defined in another binary program. This appears to be a privileged occasion to apply our coupling mechanism.

iTcl – The scripting language Tcl and its GUI library Tk [23] proved to be an efficient solution to interface an application kernel; iTcl is an object-oriented extension that helps the programmer to structure his scripts. This is an open programming environment by nature, often taken as the final application layer. The iTcl foreign office could be carried by a template offering class procedures, and every agent class would inherit from the foreign office. Parsing the data from the physical connections can be free if the format is tuned to be interpreted directly. The whole can be written in a few lines, and we hope the programmer will immediately take advantage of this standardization of coupling.

Oz – With its coherent integration of many programming paradigms, Oz [13] is a very promising language. The constraint programming model can be spectacularly powerful. The authors felt the need to add GUI facilities, both for the environment itself (the Browser) and as a development kit. They chose to rely on Tcl, but they decided to re-interface the Tcl command set in Oz; the result is not very elegant, because this new syntax is tedious for both communities. We bet that our mechanism would greatly enhance this part of Oz. Moreover, the possibility of distributing the Oz engine is being studied, and it is not impossible that the multi-country principle can be of some help. By the way, Oz is a good illustration of the obligation to keep certain values local: the identifier of an Oz object can yet be stored in a variable, but it has a protected representation that will never be cracked.

4 Programming Methodology & "Twin" Paradigm

4.1 Mechanism *vs* Methodology

Defining a general mechanism to couple heterogeneous programming environments in a coherent way is yet an important step, but another is to propose some kind of strategy to use these new facilities when designing a multi-language application. In fact, decomposing an application into a multi-agent society is quite difficult in general, and the foreign office paradigm, while offering the ability to combine the power of different languages, does not in itself indicate how to organize the agent set across the countries.

Sometimes the application contains components with a role that clearly positions them at a country. This may be the case when the behavior of this entity requires mostly the privileged skills of that environment (interactivity for Tcl [23], logical deduction for Prolog [24], number crunching for C etc.). But the situation is more delicate when the natural agent decomposition leads to some components that need different kinds of skills. In general, the choice of a particular solution will be influenced by many factors, such as the location of the main correspondents or the expected

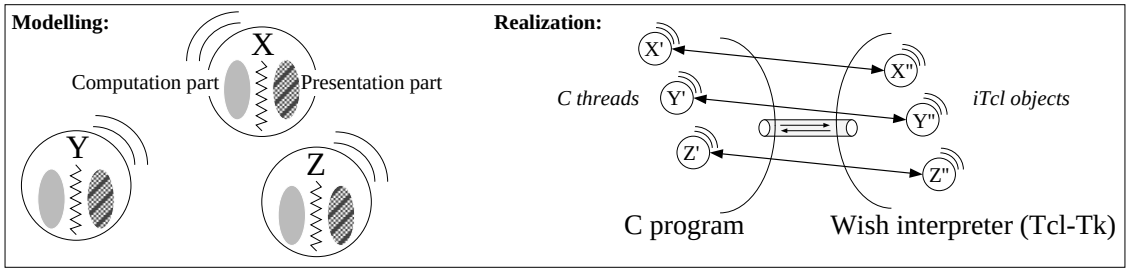


Figure 8: Twin paradigm.

Declaration	Description
`PROC SpawnTwin`	Create the twin in a remote country
`(IN Visa kind,IN Agent me)`	remote template + my local identifier
`PROC SendToTwin`	Communicate with a twin
`(IN Agent me,`	my local identifier
`IN String country,`	remote country name
`IN MsgKind msg,IN MsgArgs args)`	msg tag and data (will be translated)
`PROC GetTwinVisa`	Iterate in the twin ring
`(IN Visa twin1, IN String country,`	known twin + remote country name
`OUT Visa twin2)`	visa of the corresponding twin there

Figure 9: Foreign office extension to support the twin paradigm.

communication load. Here we introduce with the *twin* paradigm a methodology, which will help the programmer to structure a multi-country program.

4.2 Twins: Concept and Realization

The basic idea is to split the data and the behavior into two (or more) tightly-coupled sub-agents, and to write each part in the relevant programming environment. Every sub-agent is called a *twin*, in the sense that they all convey several material existence of a unique individual. This can be interpreted as a form of role delegation inside one agent.

Suppose, as in Figure 8, that there are three agents that exhibit both a computational and a presentational part. This happens for example with active results that evolve independently and must be visualized in a graphical editor, or with data interpretation experts that presents a control panel to let the user tune internal parameters. The right part of this figure shows how to apply the twin paradigm using the foreign office mechanism, to finally take advantage of C and Tcl-Tk. We could say that agent X has one foot in C (twin X') and another in Tcl (twin X").

Thus the twin paradigm can be implemented on top of the foreign office mechanism, but this implies some extra management inside the sub-agents to maintain the visa of the related twins. We therefore find it more elegant to extend the foreign office specification with the three routines described in Figure 9. The idea is to manifest the tight connection between twins: each twin visa will only differ in the prefix indicating its location (they have an identical $\beta\#\mu$). So the visa of one twin gives indirectly access to all of its siblings. Moreover, the explicit definition of the primitives `SpawnTwin` and `SendToTwin` will lead to more readable codes.

The interconnection dynamism described in Section 3.4 is even more interesting when applied in conjunction with the twin paradigm. Because the visa of one twin actually determines the others, if twin A' knows twin B' inside the same country, it can make A" know B" in another.

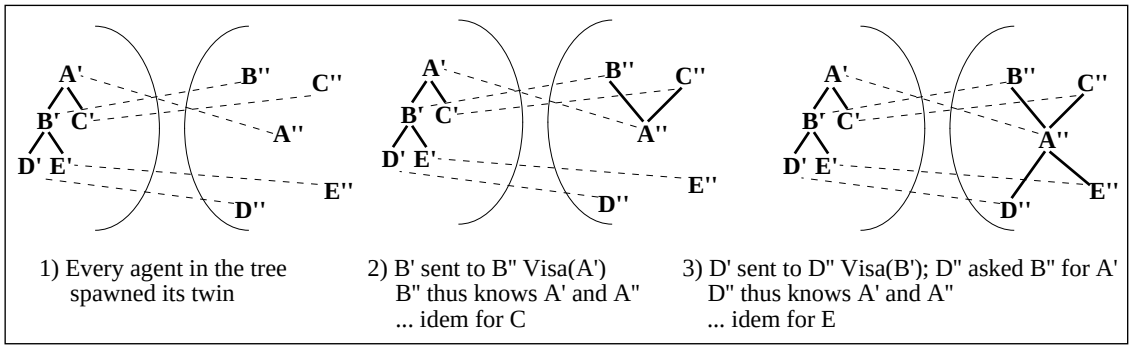


Figure 10: Two twin populations with different topologies.

Thus it is possible to rebuild the same topology among agents in two different countries, in which case both twin populations will look as through a mirror. Finally, the programmer has the ability to organize every local twin set in a different topology, as illustrated in Figure 10: suppose here that the computational twins need to be connected in a *tree*, but their user interface representative need to share a set of widgets, and therefore would prefer a *star* schema. Globally, this flexibility can bring new ideas to the designer looking for the most relevant interconnection strategy.

4.3 Some Applications

A General & Extensible Tracer – The twin paradigm is particularly suited to deal with tracing facilities. The first application is the design of a general tracer kernel that can be plugged on any system respecting our multi-agent country template. The idea is to offer an iTcl script that defines the class of a basic tracer-twin, able to present graphically the online history of every agent protocol operation. Typically, it would consist of drawing a symbol per agent, and letting the user query the past of a certain agent in terms of sent/received messages and requested spawns. There is nothing new here, but our framework freely offers some original and pleasant advantages:

- *Full monitoring*: it is relatively easy to extend the one-directional tracing functionality with a bidirectional interaction. Then, the user can take an active part in a program execution by sending messages or spawning new agents. This is a powerful and consistent testing tool, offering the human operator the core operations of an agent.
- *Extensibility towards a full application GUI*: once the class of presentational twins is available, it becomes natural to use iTcl inheritance in order to set up an application-tailored GUI. Through class derivation, the programmer can structure the end-user commands and tune the presentation of data.

We plan to evaluate the relevance of such a tool for the PT-PVM platform. It will be interesting to compare the results with the standard solution proposed by the authors of PVM: xpvm [11] is designed as a debugging and profiling means, whereas we want to reach the best level of expressiveness and re-usability.

A Real-Size Project in Document Image Analysis – The *CIDRE* [2] project deals with structured document recognition, with an emphasis on user intervention and cooperative architecture (decentralized control). There are several kinds of agents:

- Data managers: each document part (pages, blocks, lines) is managed by a dedicated C thread whose behavior drives the recognition. Typically a block first submits an OCR request, and can be activated by `add-line` messages. In fact, the system maintains one shared tree structure, where each node is managed by a separate thread.
- Specialists: each domain expertise (e.g., segmentation) is represented by a server (C thread), which is invoked by data managers. The server can master a set of workers (e.g., as separate Unix processes).
- GUI agents: each of these computational agents has a presentational counterpart encoded as an iTcl object, which handles the dialog with the user. That's where we successfully used the twin mechanism.

You can find in appendix the description of a typical scenario for this application. We found that the foreign office and twin paradigms led to the best solution to organize the relations between the concurrent C threads and GUI elements. The result is a very expressive coding: the frontier is clearly defined and the linguistic distinction prevents the programmer from writing tricky code, where GUI stuff is "disturbing" the algorithmic flow. In a certain way, adopting our approach is the most natural means to follow some conceptual models like SmallTalk's MVC or PAC [8].

5 Conclusion and Perspectives

This paper presented a generic coupling scheme to mix heterogeneous programming environments. The whole work is motivated by the following convictions:

- The need for multi-language software will increase, because subsystems are often better expressed with different programming paradigms (imperative, logic rules, object-oriented, constraints, data-flow, threads, event loop etc.). Also the reuse of existing software (legacy software) will contribute to the increasing need of heterogeneous programming environments.
- In order to propose a universal solution for multi-language management, one way is to look for some unifying programming concepts, that can be integrated in all computing models as a natural programming style, in spite of the differences in the language-dependent detailed implications.
- Multi-language discussion and coordination theory must be drawn closer together. Organizing the interactions between several heterogeneous software entities is indeed a coordination problem.
- Designing a reasonable solution requires discussing many typical examples that mix very heterogeneous programming environments. Different propositions should be assessed with respect to expressiveness, simplicity, software engineering aspects (maintenance and reuse), and consistency.

In this context, our contribution consists in sketching a simple but widely usable framework. We chose to look for some interesting analogies. The paper introduces and specifies intuitive concepts like agent, country, foreign office, visa and twin. They convey a general interconnection mechanism between coordination spaces holding active entities, typically among very different programming environments. Here are some hot topics that we are currently working on:

- a complete specification of both interface languages describing (i) spaces and (ii) agent kinds;

- the development of source code generators, able to produce a whole carcass for a multi-language application specified through the interface languages;
- a powerful improvement of our model, to allow the universal use of `OUT/INOUT` parameters in message passing;
- the hiding of visa management with the definition of "proxy agents", i.e. system generated local "proxy" agent of a remote agent, to allow the transparent use of local protocols for `spawn/send` operations.

The promising experiences made so far with Oz [13], PT-PVM [17] or Tcl [23] encourage us to validate deeper our new coordination facility with various environments and on real-size applications (like those evoked in Section 4.3). We believe that the proposed architecture can be of some help for any programmer wanting to take advantage of several languages in a single application. Moreover, our approach could help the resolution of some non-trivial coordination problems (e.g. designing a distributed version of Oz). Globally, this contribution opens new perspectives around the design of software architectures.

Acknowledgement. We would like to express our gratitute towards Prof. Béat Hirsbrunner and Prof. Rolf Ingold, who encouraged us to intensify the collaboration between our research groups. This paper is a concrete result of the exciting synergies that we discovered between parallel computing and document recognition. We also want to thank Christian Renevey for his participation in the first drafts.

A Typical Scenario of a Twin-Based Application

Following is the discussion of a typical scenario for the *CIDRE* prototype [2]. We want to show that our coupling mechanism is transparent and that the twin paradigm seems here very natural.

Figure 11 sketches the evolution of the system during the analysis of a new document page. Every agent behavior is represented by a vertical thread showing its state transitions. Interactions are either agent spawning or message passing.

When the new page is created, the corresponding data agent generates its presentational twin, and submits a request to the segmenter. This specialist finds one of its remote worker to perform the image analysis; when the job is over, the segmenter translates the results in terms of a sequence of `add-child` messages. This re-activates the computational page twin which has now to merge these new block sub-trees with its existing children (if any!). If no matching is found, the new entity is accepted and a block agent is spawned, which solicits the OCR in an analogous way (the OCR worker is not represented in the figure).

The interesting point is that `add-child` messages can be issued from different sources, notably from the user (via the presentational twin). In general the user is free to give his own interpretation of the data at any time, e.g. to set the font of the block before the analyzer is called on every line. When a new result causes a conflict (here the block faces two incompatible line decomposition), the data agent can invoke a revision of the former interpretation, or catch the operator's attention through the GUI.

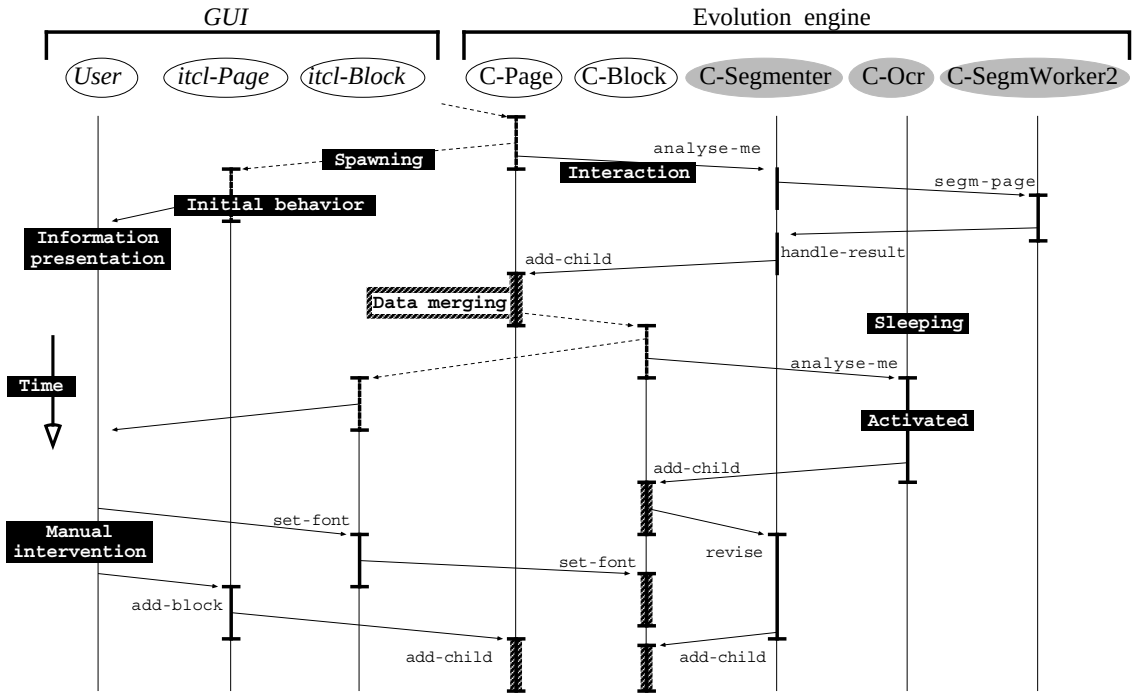


Figure 11: Evolution of a typical scenario.

References

[1] Object Request Broker Architecture. Technical Report OMG TC Document 93.7.2. Technical report, Object Management Group, 1993.

[2] Frédéric Bapst, Rolf Brugger, Abdelwahab Zramdini, and Rolf Ingold. Integrated Multi-Agent Architecture for Assisted Document Recognition. In *DAS'96*, Malvern, Pennsylvania, October 1996.

[3] J. G. P. Barnes. *Programming in Ada.* Addison-Wesley, 1984.

[4] O. Baujard, S. Pesty, and C. Garbay. MAPS: a Language for Multi-Agent System Design. *Expert systems*, 11(2):89–98, May 1994.

[5] J.A. Bergstra and P. Klint. The ToolBus Coordination Architecture. In Paolo Ciancarini and Chris Hankin, editors, *First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.

[6] M. Bever, K. Geihs, L. Heuser, M. Mulhauser, and A. Schill. Distributed Systems, OSF DCE and Beyond. In A. Schill, editor, *International DCE Workshop*, number 731 in LNCS, Karlsruhe, October 7–8 1993. Springer Verlag.

[7] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[8] Joëlle Coutaz. *Interfaces Homme-Ordinateur.* Dunod informatique, 1990.

[9] David Flanagan. *Java in a Nutshell.* O'Reilly, 1996.

[10] I. Foster and S. Taylor. *Strand - New Concepts in Parallel Programming.* Prentice Hall, 1990.

[11] Al Geist, Adam Bueguelin, Weicheng Jiang, Robert Manchek, and Vaidy Sundeam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, Massachusetts, 1994.

[12] W. Gora. *Abstract Syntax notation one.* DATACOM, 1992.

[13] Martin Henz, Gert Smolka, and Jorg Wuertz. Oz- A Programming Language for Multi-agent Systems'. In *Internationl Joint Conference on Artificial Intelligence*, pages 404–409, Chambery, France, 1993. IJCA-1993.

[14] G. W. Johnson. *Labview Graphical Programming: Practical Applications in Instrumentation and Control*. McGraw Hill, 1994.

[15] Astrid M. Julienne and Brian Holz. *ToolTalk and Open Protocols: Inter-Application Communication*. SunSoft Press Englewood Cliffs, 1994.

[16] O. Krone and M. Aguilar. Bridging the Gap: A Generic Distributed Hierarchical Coordination Model for Massively Parallel Systems. In *Proceedings of the '95 SIPAR-Workshop on Parallel and Distributed Computing*, Biel, Switzerland, October 1995.

[17] Oliver Krone, Béat Hirsbrunner, and Vaidy Sunderam. PT-PVM+: A Portable Platform for Multi-threaded Coordination Languages . *Calculateurs Parallèls*, 8(2):167–182, 1996.

[18] H. Laasri and B. Maitre. Flexibility and Efficiency in Blackboard Systems: Studies and Achievements in ATOME. In Academic Press, editor, *Blackboards and Applications*. 1989.

[19] Sun Microsystems. *XDR: External Data Representation Standard*. Sun Microsystems, 1987.

[20] F. Mondada, E. Franzi, and P. Ienne. Mobile Robot Miniaturization: a Tool for Investigation in Control Algorithms. In *ISER'93*, Kyoto, October 1993.

[21] Adrian Nye. *Xlib Programming Manual*. O'Reilly, 1990.

[22] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1996.

[23] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesely, 1994.

[24] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[25] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

Contents